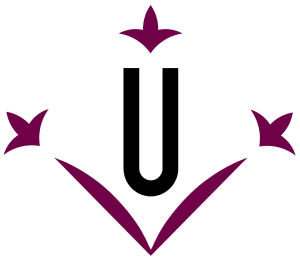
Universitat de Lleida
Escola Politècnica Superior
Màster en Enginyeria Informàtica



Universitat de Lleida

Mechanical Dystopia: A roguelite videogame developed in Unity

Sergi Vila Almenara
Director: Francesc Sebé Feixas

Treball Final de Màster, Juliol 2017

Acknowledgements

To my family, friends and office mates for supporting me

To my tutor Francesc for guiding me in this project

To all authors of the used resources

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Some people think that developing video games is a fun task, just as playing them is. The truth is that they are right, but just to a certain extent. Creating video games requires a wide variety of knowledge and criteria, so making a game is not far from trying to develop a banking application, a personal web or setting different interactions with a database.

The different human crafts can be divided in different classes of arts: painting, cinema, music, etc. Among them, videogames have the virtue of being able to merge most of this arts at the same time, offering an experience that can possibly surpass what any of them can achieve individually, if the mix is the adequate.

Every videogame has some specific but common parts: a story (which can be told through text and/or cinematic media), a soundtrack, a design through some kind of artistic style and a bunch of playable mechanics. Meanwhile the rest of the arts are enjoyed passively, meaning that customer is a simple witness, in a video game, the subject is indeed the protagonist and the engine of action. With a greater or lesser degree of freedom for every game, there is a set of actions implemented, that are previously designed by the author, to be used as tools in order to finish the game. Therefore, the player executes the actions and uses those tools, it is giving a real (though fictitious) challenge to complete.

My journey in game development began about nine years ago, at my age of 14, when without much idea I was manipulating and trying to understand programs like *GameMaker* [75] and *RPG Maker* [19]. The projects that were started using these tools were abandoned due to a lack of perseverance, however, must be said, that this programs were not difficult to use or manage to develop things with.

After school and compulsory education, I started a Degree in Computer Engineering at the Universitat de Lleida. After having learned to program minimally, I started to investigate about Unity, which was still on its 3rd version, and was still pretty basic back then. I was able to follow some tutorials and perform very basic tests that altogether seemed to work.

In addition to my studies, I performed company practices and extracurricular practices in *Plunge Interactive* [35], a video game company based in Lleida, mainly dedicated to the outsourcing of smartphone games. In it I was able to learn from the inner workings of some already developed games, Tiny Troopers 1 [65] and Tiny Troopers 2 [66]. Thanks to all experience gained, I improved my abilities and skills in Unity and I was able to start developing my own videogame projects.

Since then, I have been carrying out the following projects:

- The Game of Life,
- Endless runner compatible with virtual reality,
- Applications based on lists,
- Online image viewer,
- *Guitar hero* [3] clone using an *Arduino* [9] board,
- Various musical experiments using Arduino,
- Customizable *Tetris* [11],
- A lot of tutorials and small projects to test and learn new features.

After finishing the degree, I started a Master in Computer Engineering with a specialization in video games at the Universitat de Lleida and the project that the reader is able to experience right now is the final step of it. During this master, I learned the basics of graphic programming and video game logic, using *OpenGL* [32] I developed a *Pacman* 3D game and a 2D pool. In addition to that, I developed a 3D tank game and a 2D platform game using *Unreal Engine 4* [30].

Having accomplished all these challenges and being able to end up successful in most of them, the natural step to continue my learnings was to carry out a project of larger proportions.

This document describes how Mechanical Dystopia was originated with its timing and costs (Chapter 1). The reader is introduced to the tools used during development and roguelite games in (Chapter 2). The design document of Mechanical Dystopia is presented in (Section 3.2), its user-level operation in (Section 3.4) and the implementation of each of its functionalities in (Section 3.5). Finally, in the

conclusions section, there is an assessment and discussion of the content that could be expanded or improved and a meditation and afterthought about what is developing a videogame per se, based on the experience obtained in this thesis (Chapter 5). Annexes are also attached for further information.

1.1 Motivation and Objectives

Since 5 years ago the roguelite genre attracts me, first with several games such as The Binding of Isaac, Spelunky, Legend of Dungeon, Nuclear Throne, The Binding of Isaac: Rebirth, Risk of Rain, FTL: Faster Than Light, Sunless Sea, Luftrausers, Crypt of the Necrodancer, Downwell, Enter the Gungeon, Teleglitch among many others.

The idea for a game arises from the graphical effects used in some videogames. For example, the musical levels of Rayman Legends [25] include modifications on the entire screen, but the rest of the game does not include them.

Other games modify the aspect of the scenarios and characters applying shaders to their materials. For example, a dissolution shader [53], showing the silhouette of the character behind objects [49], revealing hidden platforms [5], or curving the scenario [4].

Finally, some games focus part of the gameplay on effects, most of them scary or terror games, like Amnesia: The Dark Descent or Eternal Darkness. If the character feels threatened, he will lose his sanity, see non-existent enemies, sounds, movement on inanimate objects. Here comes the idea to apply graphic effects that evolve over time based on player decisions, not only as part of the level design.

Once the type of game to be done has been decided, the main objectives are:

- Evaluation of the current indie videogame industry and its business model
- Creation of a roguelite videogame using a high-tier game engine
- Implementation of original game mechanics
- Design of 3D models including rigging, texturing and animations
- Learn new technologies currently used in game development

1.2 Timing

The project formally began on October 20th, 2016 after the completion of a draft version of its the design. The initial scope of the project was moderated. The *Gantt diagram* presented in Figure 1.1 shows an initial expected timing:

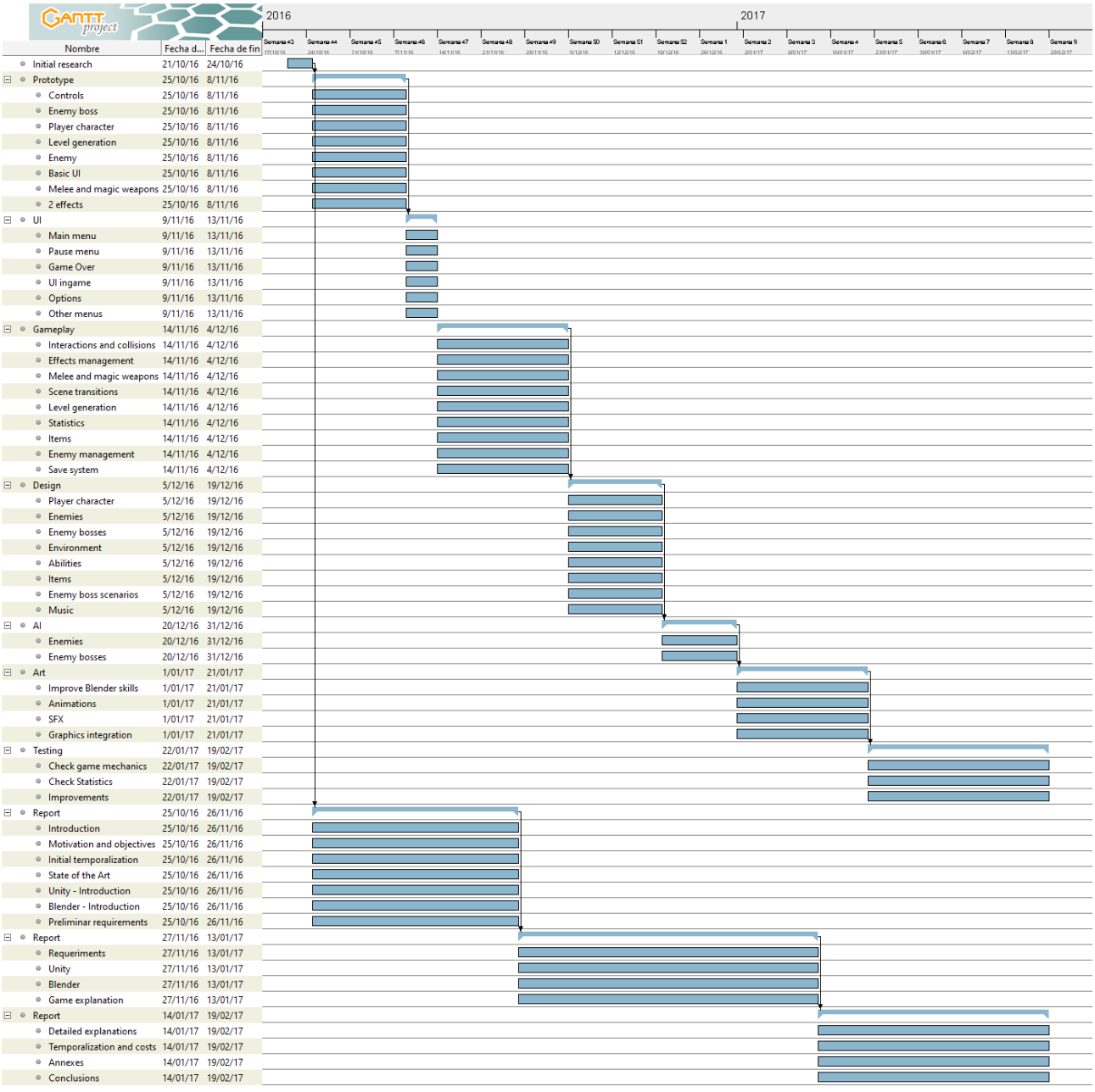


Figure 1.1: Inital schedule

The project size has been growing over the months, for beyond the initial objectives. Also, during some periods, the project has been paused due more priority academical tasks. The following diagram (based on commits) offers a more realistic timing:

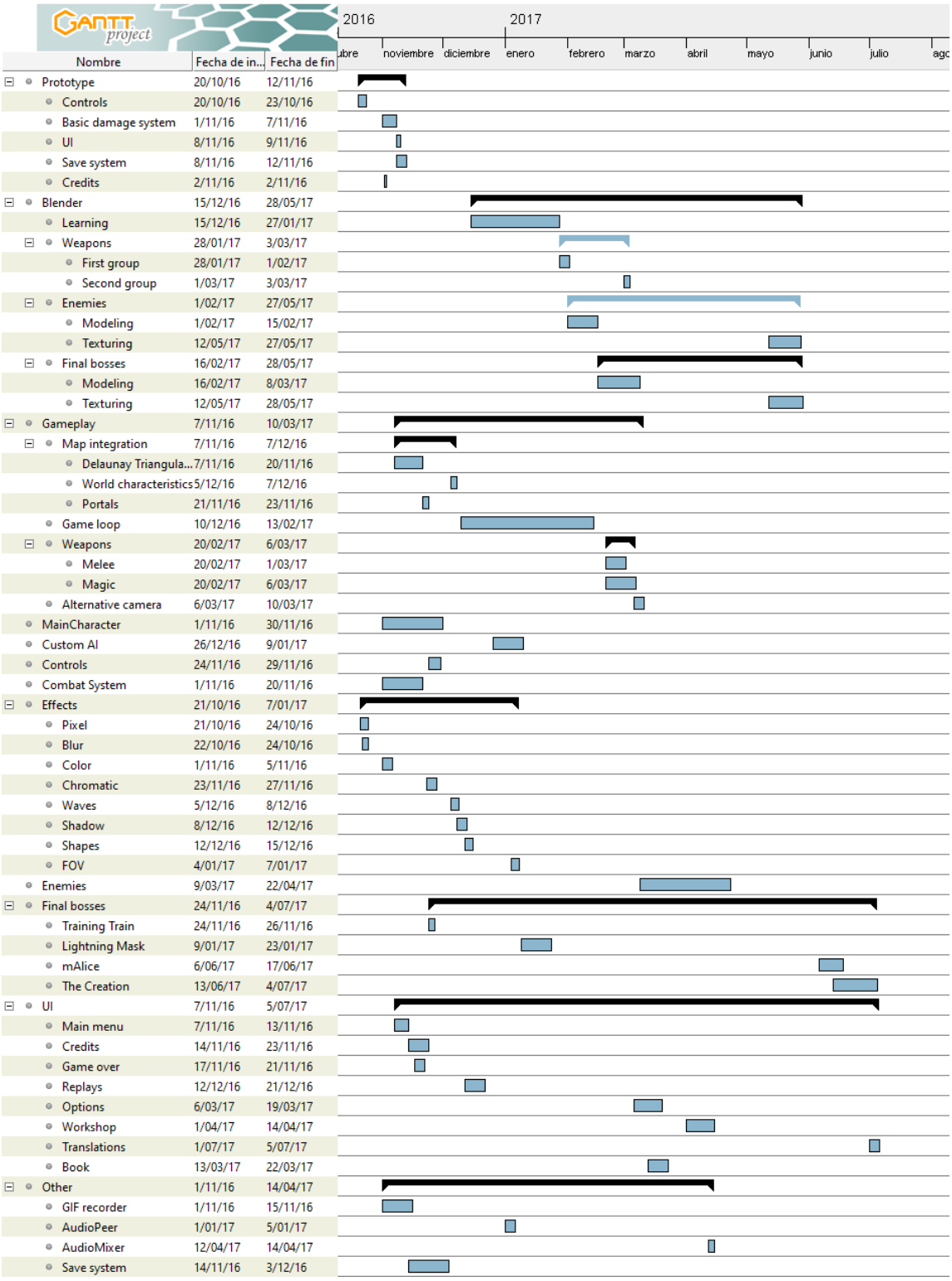


Figure 1.2: Final temporalization

1.3 Costs

Due to it being an amateur project carried out by one person, the cost of the project is near 0. But some hardware and software were required to develop the project.

The used hardware was the following:

- Desktop PC with Intel i7, Nvidia GeForce GTX 560 Ti and 8GB RAM + peripherals
- Laptop PC with AMD Quad-Code A10-7300, AMD Radeon R6 and 8GB RAM + peripherals
- Graphic tablet XP-Pen Star 03
- E2C instance from Amazon Web Services

The used software was the next:

- Windows 10
- Unity
- Visual Studio 2015 Community Edition
- Sublime Text 2
- Overleaf
- GIMP
- Adobe Photoshop CC 2015
- Blender
- SourceTree
- Amazon Web Services

This game was developed mainly at home, with some parts implemented in EPS offices. Finally, my salary is probably the unique expensive cost, but in a real case, the salary is all the benefit of the sells obtained by the game, so there isn't a fixed value.

After these considerations, the overall budget of the project is:

Description	Amount (€)
Hardware	
E2C instance	Free the first year
Routing of E2C instance	1.00 per month x 6 = 6.00
Web domain	1.00
Graphic tablet XP-Pen Star 03	57.00
Software	
Unity license	Free
Visual Studio 2015 Community Edition	Free
Adobe Creative Cloud subscription	12.09 per month x 6 = 72.54
Blender	Free
SourceTree free plan	Free
GIMP	Free
Windows 10	Free upgrade from Windows 8.1
Sublime Text 2	Free trial
Unity Asset Store	
Procedural Gear	4.63
Art	
Graphical resources	0.00
Music	0.00
SFX	0.00
Salary	15.00 per hour x 400 = 6000.00
Total	6141.17

Chapter 2

State of the art

2.1 Introduction

In this section we present the tools used in the project and a little demonstration of a Unity project. We also present to comment some concepts that will be named throughout in the document, all the necessary explanations to understand how roguelite videogames work, and finally, how the indie industry works and my particular vision about indie videogames monetization.

2.2 Unity

Unity [52] was created in 2004 by Over the Edge I/S (currently Unity Technologies SF) as a game engine for the game GooBall. Sadly the game did not succeed, but developers realized they had created a complete all-in-one general-purpose solution that could be licensed to other game studios or even to the general public.

The most remarkable games done with Unity include: Aragami, Yooka-Laylee, Ori and the blind forest, Dreamfall Chapters: The Longest Journey, Pokemon GO, Hearthstone: Heroes of Warcraft.

2.2.1 Features

Unity works similarly to other visual game engines. The programmer can drag & drop content from the project folders to the editor, set properties to the objects, combine them and write scripts to

create behavior and logic. It includes all the necessary tools to develop any kind of game.

- Components: Any script that extends the class *Monobehaviour* is a component. This is an attachable piece of code that expands the functionalities of the receiver object. There are other types of components like meshes and render settings, layout configurations, motion and materials with properties for physics interaction, audio, navigation among many others.
- Unity Asset Store: The store of Unity is full of content like animated 3D models, environments, music, particle systems, scripts, even complete projects. Most of the content requires a payment, but there is also a lot of free content.
- Platforms: Much of the success of Unity lies in its broad device compatibility, being the most remarkable: Android, iOS, PC, PlayStation family, Xbox family, Wii U, 3DS, Switch and virtual reality.
- Windows: Unity is a graphical tool, not like other engines that are purely code libraries.
- Ease of learning and use: Unity offers some ways to program games even without requiring previous knowledge on programming. Following some tutorials and using visual assistants, basic games based on physics can be done very quickly.
- Compatibility: Practically any resource can be imported by Unity: images, videos, audios, 3D models, compiled libraries and resources from other projects. They are editable by external programs without requiring an explicit reimportation.
- Advanced tools: At each update, Unity includes additional technologies to accelerate game development, e.g., a new UI system, light mapping, reduction of draw calls using some integrated techniques, a profiler, new options in AI, navigation, terrains and networking.

2.2.2 Windows

- Scene: It is the view of a floating camera to view and create levels. Selected objects (called *GameObject*), can be moved, rotated and scaled. It can also be used in runtime to check unseen zones of the game camera or edit objects.
- Game: The view of the main camera used in runtime.
- Console: Shows log messages, warnings and errors.
- Hierarchy: Displays all the objects in the scene. Dragging an object to the Project window will create a prefab, which is a template object that can be instantiated. Multiple scenes can be managed at the same time.

- Project: It is a folder manager to organize the assets and scripts of the project.
- Inspector: Manages the components attached to a *GameObject*. All the scripts that extend *Monobehaviour* can be attached to a *GameObject*.
- Animation: This graphical tool is used to configure an animation clip.
- Animator: This is a configuration window to create state machines for animations.
- Audio Mixer: Permits the configuration of sound groups (music, effects, voices, UI) and the interactions among them.
- Profiler: Graphical profiling to analyze the performance of the game. At grain level it can show the resource usage of each function.

Unity also provides tools to create new windows, menus and integrated contextual menus. For example, *Shader Forge* [33] allows the creation of materials in a similar way *Unreal Engine 4* does, using a graphical windows for connecting nodes.

2.2.3 RAIN AI

Development of artificial intelligence for enemies could be a big issue, since the navigation system must work according to these logics. The big deal found during the development is that the *NavMesh* system provided by Unity works at compilation time, that is, dynamic environments can't use it[1]. The best alternative found to solve this issue is *RAIN AI* [64], created by *Rival Theory*. It is a powerful AI module that allows Unity developers to implement the behavior of characters. The main features are a new *NavMesh* system compatible with patrol paths, a visual behavior system, *Mecanim* integration, and different types of sensors, all compatible and customizable using C# scripting. Sadly, there are only some examples and video tutorials. Most of the documentation has not been done yet. This fact it made difficult to learn of some of its features. It is worth mentioning the simplicity of usage of the *NavMesh* and how the characters move realistically according to him.

2.2.4 Methodology and workflow

Pure-code game engines requires the completion of the entire game using external tools for most of its parts like 3D models, environments, particle systems, animations, AI, textures and UI graphics and music, relegating the engine to just merge all these data. With Unity these tasks can be done inside

[1]Since version 5.6 released on March 31, 2017 the generation of NavMesh in runtime is supported

the engine, either those integrated tools or by using plugins. This allows faster developments, less specialization and a perfect compatibility.

In Unity, the drag & drop functionality and the inspector are probably the best features for newbie game developers, offering a low difficulty curve. This is why Unity is a good engine to start instead of other engines that are more complex or oriented to high budget games.

A proposal of windows layout in Unity (used in the project) is next shown:

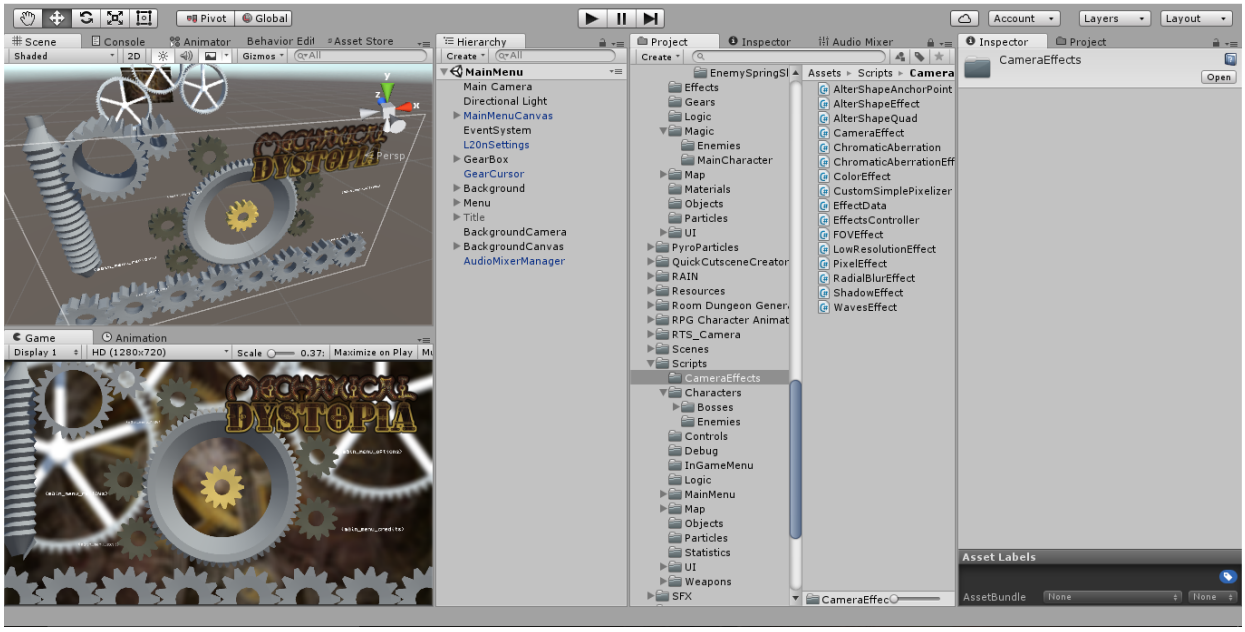


Figure 2.1: Proposed Unity layout

2.2.5 Simple example

Now will show how to create a cube spawner to familiarize with the development environment.

We start the project creating "Prefabs", "Scenes" and "Scripts" folder using the "Project" menu and saving the current scene with File >Save Scene in the Scenes folder (Figure 2.2). Then, we are going to create a floor selecting *GameObject* >3D object >Plane. Plane object is added and it will be displayed in *Scene* and *Game* screens.

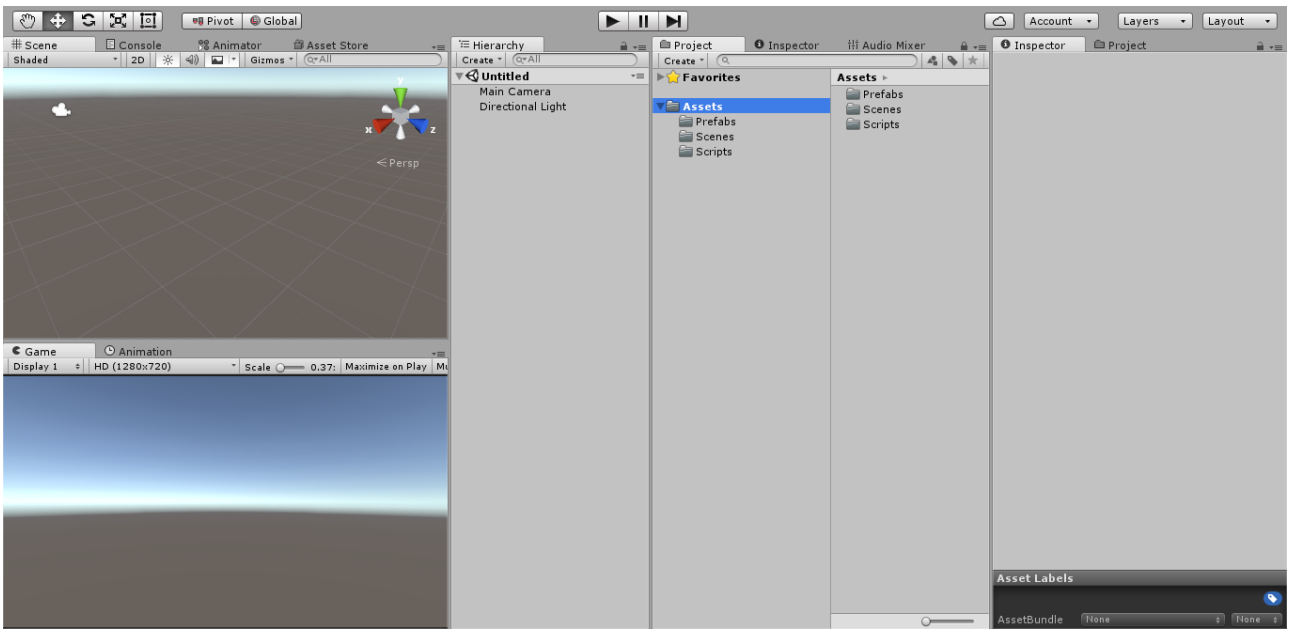


Figure 2.2: Starting a cube spawner project

Now we will add a cube with *GameObject* >3D object >Cube. In the Inspector there is the Transform component to set the position, rotation and scale. There is also a Mesh Filter with the cube model, a *Box Collider* to detect collisions with other colliders, a *Mesh Renderer* to set how the model is displayed (e.g. materials, shadows) and finally the shader used in the materials. Using the button "Add Component" we add a *Rigidbody* component (Physics >Rigidbody). Now the cube is a Physics object, being affected by gravity, impulses and shocks. This cube is a specific object in the scene, but we want it to became a prefab. By dragging it from Hierarchy to the Prefabs folder in Project, the cube name become blue. Now it can be used in other places. We can delete the cube in the scene. The next step is the creation of an empty object (GameObject >Empty). We can select one of the *gizmos* by clicking in the blue, red and green cube at the left of the name of the object in the inspector. Also we can change the name to "CubeSpawn". We move now the object to the (0, 1.5, 0) position. With a right-click in the Scripts folder we select Create >C# script to add a new script to the project. We open the file in *Monodevelop*, *Visual Studio* or another *IDE* and paste the next code:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class CubeSpawn : MonoBehaviour {
5
6      public GameObject cubePrefab;
7      public float spawnTime;
8
9      float currentTime = 0f;
```

```
10
11    // Use this for initialization
12    void Start () {
13
14    }
15
16    // Update is called once per frame
17    void Update () { // Every spawnTime seconds a new cube will be instantiated
18        if(currentTime < spawnTime)
19          {
20              currentTime += Time.deltaTime;
21          } else
22          {
23              currentTime = 0f;
24              SpawnCube();
25          }
26    }
27
28      void SpawnCube()
29      {
30          GameObject cubeInstance = Instantiate(cubePrefab, transform.position,
            ↪ transform.rotation) as GameObject; // A copy of cubePrefab at the
            ↪ position of CubeSpawn object is generated
31          Vector3 impulse = new Vector3(Random.Range(-300f, 300f), Random.Range(300f,
            ↪ 600f), Random.Range(-300f, 300f));
32          cubeInstance.GetComponent<Rigidbody>().AddForce(impulse); // The cube is
            ↪ powered up with some inclination
33      }
34  }
```

Selecting the *CubeSpawn*, we add this script (Scripts >Cube Spawn). There are two variables to be set: Cube Prefab (click on the circle at the right and select Cube), and the spawn time, a decimal number, e.g. 0.5. Clicking the play button the game starts and we can see how the cubes are generated. Selecting the Cube Spawn we can modify at runtime the spawn time value (click the variable name and move the mouse to the left and to the right). After stopping the game the variable takes to its original value. Going to File >Build settings... and clicking Add Open Scenes, pressing Build and Run and writing a name in the next window, the compilation of the project starts and we can see our game running on our PC (Figure 2.3).

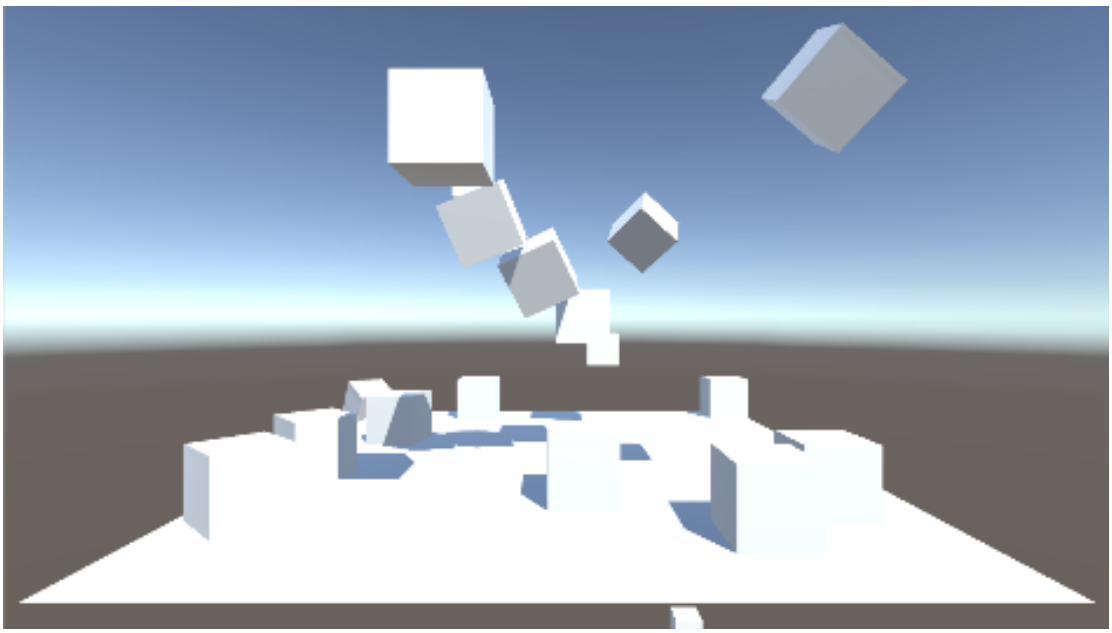


Figure 2.3: Cube spawner working

This basic project showed us how to use some of the menus, create new objects, add components, write scripts, use Physics and deploy the project.

2.3 Art tools

Any videogame requires art for all the objects that can be seen in the game must be created (environments, characters and objects). Depending of the graphics style (e.g. realistic, cartoon, minimalistic, pixel art, hand-drawn) one or more programs appropriate their creation are needed.

2.3.1 2D

Photoshop

Photoshop was created in 1988 by Adobe Systems Incorporated. It is probably the most known and used photo editing tool. It is suitable for a wide variety of tasks like photo retouching, color correction, photo restoration, draw by hand using a graphic tablet and design of any kind of art like advertising material, realistic painting or pixel art.

GIMP

The open source and free alternative to Photoshop is GIMP, created in 1996 by Peter Mattis and Spencer Kimball. It allows to perform practically the same actions (by adding plugins) as the paying competitor with slight differences. GIMP is less powerful but it requires less resources.

Comparison

In my opinion, GIMP and Photoshop are similar and both include all the required functionalities for an amateur artist. A positive point for GIMP is due to its open source slope. The difficulty curve for Photoshop is lower, the position of buttons, windows and menus, the hotkeys, the use of the tools and other subtle details are more natural.

2.3.2 3D

Blender

Blender[24] is a general purpose 3D design application for modeling, sculpting, creating animations, texturization, creation of scenes and video, rendering, simulating physics and much other features available by means of plugins. It is available in Windows, Max OSX and GNU/Linux.

The main features of Blender are:

- Modeling: The core of Blender is the creation of 3D models from the manipulation of vertices, edges and faces.
- Rigging: Once a model is finished it is just a static mesh. We need to provide bones to the model to animate it. Not all bones are used as a skeleton. There are special properties that can be attached to external bones to provide features like head and hands pointing, detailed facial expressions, feet over floor (useful when a character is over irregular surfaces like stairs) or better join rotations.
- Animating: Once the bones and weights have been configured, animations can be created from keyframes by positioning and rotating the bones in the desired place.

- Texturing: The unwrapping technique allows to deploy each face of a model in a flat representation. From this window a texture can be added to Blender, integrating it into the model. The most common technique is to export the faces pattern for use in a photo editing program, and finally import the final texture in Blender. There are different ways to paint a model, e.g. painting directly on the model, or painting and setting all the textures directly in Blender.

Alternatives

There exist great applications for 3D design, mainly used to generate post-production effects for movies (hence, cinematics in video games).They can even create behaviors and physics to automate scenes: the movement of branches and leaves caused by the wind direction, displacement of large groups of characters and their individual behavior depending on the situation, physical fluids and shocks between elements.

Next, we present some of them:

- 3DS Max: It is very similar to Blender but requires a expensive subscription, its owner is *Autodesk*. It is oriented to architecture and modeling.
- Maya: Created by *Alias System Corporation* in 1998 and bought by *Autodesk* in 2004. It is also similar to *Blender* and *3DS Max*. It requires a subscription but it is more flexible for testing it and it is free for students. It is oriented to animations.
- Cinema 4D: It was created by *Maxon* in 1980. It has the same capacities as the previously cited, but provides better results for flat media like static scenes, images and motion graphics (cartoons).
- ZBrush: Specially designed to sculpt and paint organic models.

Following the same parallelism as *Photoshop* and *GIMP*, *Blender* is enough powerful for small projects, even half productions. Its possibilities are broad and allows to generate high quality content, although its difficulty curve is greater due to its interface.

2.4 Roguelite videogames

The roguelike genre was originated by *Rogue* [39][44][10], a videogame designed by Glenn Wichman and Michael Toy in 1980. The player is an adventurer that must find an amulet in the last basement

of a dungeon and return to the exit. The game is executed directly on a console terminal as it can be seen in Figure 2.4, so all the elements are ASCII characters. The player is a face (or a @), the enemies are other alphabetic characters, items are punctuation signs, the and map is composed by dots, sticks and patterns. In each level the player explores rooms with enemies and items, deciding which is the best action at each turn: attack the enemies or avoid them, pick an item but fight against enemies, go to the next level or continue exploring the current level and so on. The player gains experience, items and gold from enemies and treasures to improve its health, strength and armor.

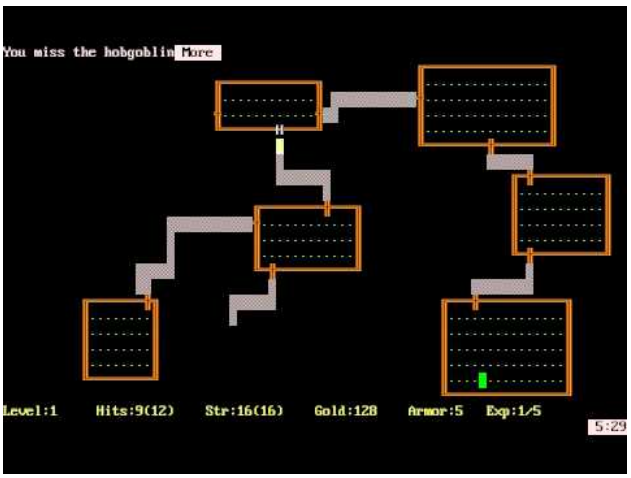


Figure 2.4: Rogue (1980)

The characteristics of *Rogue* and the roguelike genre are:

- Permadeath: Once the player dies, all the progress is lost, he must start again the entire game from scratch. The unique gain of the player is the personal experience and knowledge he has been able to capture and synthesize.
- Randomness: The map, enemies and items are totally randomized at each level. Two games are never the same.
- Turn-based role: Every player action (move, attack, use an object) is considered a turn. Once the player finishes his action, the enemies do one.
- Tilemap: *Rogue* works over a terminal, so each character is a tile. The characters are represented as symbols.
- Hard difficulty: This type of games requires a deep knowledge about the mechanics, requiring a long-term learning, recording the behavior of each element. The knowledge is usually compiled in wikis.
- Character evolution: Character starts with basic stats and equipment. The user can choose to

get more items, depending how the character will be oriented (creating a build) or searching specific combinations. If the player is lucky, the game becomes easier, otherwise, a restart will be the best option, or the character will die irreparably.

- Decisions: Due to randomness, the player will face new situations at each game. For example: Can I continue exploring the current level to obtain better equipment, or I continue to the next level?; I can open this chest, but it could contain an enemy; I have the option to get a large-range weapon but its damage is less than my current short-range weapon, should I take it?.

In the last few years some games appeared redefining the roguelike genre. They are called roguelite due to the non compliment of some of the original rules.

Together to the roguelike genre, the game has some sub-genres:

- Top-down: Basically the game uses a top camera.
- Dual-stick shooter: With the left stick the character is moved, with the right, it shots.
- Dungeon crawler: The character advances though a dungeon, normally split in rooms.

Not all the roguelite games include these genres, but most of them do.

Annex A shows a complete chart about the characteristics of the most relevant roguelite videogames in PC. Most of the data has been obtained from *Steam*[57], *IndieDB*[15], *SteamDB*[74] analysis and the wikis of each game, only taking into account the base game without additional downloable content. The data presented can change along time or could be imprecise if the original font is not correct, so it must be taken with caution and only for guidance about the available content. For example, the price is a very volatile value, so we included the highest price found at *SteamDB*.

The next three games are analyzed more deeply for its importance in the roguelite genre and its influence on *Mechanical Dystopia*.

Figure 2.5: The Binding of Isaac: Rebirth

Figure 2.6: Nuclear Throne

Figure 2.7: Enter the Gungeon

The Binding of Isaac / Rebirth

The Binding of Isaac[43] was originally a *Flash* game created by Edmund McMillen in 2011. The game starts when Isaac's mother gets God orders to her to sacrifice her son, which causes Isaac to hide in the basement of his house. Isaac must advance through the rooms in each level until the boss room is found. During the game the player will fight against all kind of insects, monsters, biblical characters and his mother as one of the final bosses. Isaac can pick objects to upgrade his stats, special objects to perform special attacks and effects, get money to be spent in shops, keys to open rooms and bombs to destroy obstacles and kill enemies. All the rooms of the game are hand-made but the composition of each level is random. There are special rooms such as free item, sacrifice, hell, heaven, shop, hidden, with special events on it. Finally, not only Isaac is playable, there are unlockable characters with other stats and special abilities.

The player attacks by throwing tears, which is a slow and inaccurate attack, so player must move itself to the same time to direct the tears to the desired direction. Picking items will be alter the tears, obtaining rich accumulative combinations. For example, big poison explosive tears, some types of lasers, companions with its own attacks, change tears by other objects, etc. Probably the best feature of the game is the huge amount of combinations that can be obtained, not only to upgrade the stats but also to change the physical appearance of Isaac.

In 2012 a new expansion was launched, *Wrath of the lamb*, with new types of rooms, enemies, items and bosses. In 2014 a remake was created, *The binding of Isaac: Rebirth*. Two more expansions appeared in 2015 (*Afterbirth*) and 2017 (*Afterbirth †*).

Nuclear Throne

Wasteland Kings[70] was created by Vlambeer during a gamejam called MOJAM charity build in 2013. It was developed in only 3 days. During more than two years in *Early Access* in Steam the project becomes *Nuclear Throne*[68]. As described by its creators, "Nuclear Throne is Vlambeers latest action roguelike-like about mutants that spend their workdays trying to fight for the throne in a post-apocalyptic world. The radioactive waste in the world allows mutants to get ahead by mutating new limbs on the fly and the abundant availability of powerful weaponry makes the quest to become the ruler of the Wasteland one frought with peril.". The goal of the player is to reach the Nuclear Throne, advancing through randomly generated levels with random enemy and chests spawns. There

is a lot of mutants to choose, everyone with a special ability, e.g. roll, shield, two weapons at the same time or eat weapons to earn health and ammo.

The gameplay is frenetic, the reflexes are important to manage the high quantity of enemies and projectiles, but patience is also required for some kind of level structures and enemy behaviors. Player starts with a gun, but in the chests distributed in the level new weapons can be found, classified in bullet (rifles), shell (shotgun), bolt (crossbows), explosive (bazookas), energy (laser) and melee (wrench). Every weapon requires some type of ammo (except melee) so the player must search for ammo depending on his weapons, because he can only carry two, and can use only one at the same time. The enemies are bandits, anthropomorphic and normal animals, robots and slime monsters. There are three bosses that during the levels. The way the characters are upgraded is by collecting nuclear bars from the dead enemies. When a number of bars is obtained, the character gains a level and can choose a mutation after the completion of a level. The mutations consist on improvements like more health, destroy walls, enemy contact damages, no explosion damage if health is less than 4, more chest spawn and so on. The player must decide which combinations of weapons, mutations and character ability give more advantages depending on the situation, but luck is also important.

Enter the Gungeon

The plot of *Enter the Gungeon*[50] is that a castle called *Gungeon* appeare. In the last floor there is a weapon that "can kill the past", and our "gungeoneers" want it. Most of the enemies are bullets, but there are also animals, fantastic beasts and magicians. The player must find the boss room to reach the next floor. The rooms are hand-designed but are randomly connected. On each floor there is a shop to buy new items and weapons in exchange of money, that is obtained killing enemies. The weapons also can be found in chest, but most of them require a key, that can be bought in shops and randomly after a room is completed. All the weapons are stored so player can carry them, but only one can be used at a time. Every character has different stats, a starting weapon and an ability. For example, the pilot has less precision and starts with a basic gun, but he has discounts in shops, a lock pit for chests, an additional slot for items and more max ammo. The characters can roll, overturn tables to cover and use an item called armor to destroy all the dangers in the screen and also acts as a shield before spending health.

The star feature of the game is the high amount of weapons available, being most of them very original,

e.g. a T-shirt thrower, a bullet that shots pistols that already shots bullets, an axe-gun that performs a melee attack during recharging, the gun from Ghostbusters, a gun that throws letters. After beating a boss, special money is obtained to unlock weapons in the game.

2.5 Developers

Videogame industry is suffering a constant evolution along the years. Initially videogames were developed by few people due to they being very simple. Until '90 this was the topic. At around 2005 (Xbox 360 and PS3 generation) high-top videogames got extremely expensive to develop, requiring new technologies to improve the visual quality and the capabilities of the games.

Currently indie games are an important part of the videogame sector.Normally indie developers have risky ideas that a consolidated study with a high budget can not afford because the failure of the project may involve closing the company. Every year thousands of indie games appear and only a little part of them succeed in critics and economically. The rest of developers must develop another game and try again.

2.5.1 Barcelona Games World

From 2013 to 2015 Madrid Games Week was one of the most important videogame events in Spain. This game show is specialized on offering imminent video game releases, exclusive demos of games on development, e-sports competitions, merchandising, developers' talks and meetings with game developers. But in 2016, the game show moved to Barcelona, becoming Barcelona Games World. On October 9th I attended it with some friends.

Barcelona Games World[16] was divided into different areas:

- Expo: New games available to be played by people.
- Merchandising: Shop products related to videogames.
- Professional areas: Where professionals do conferences to people and meet among them for new agreements and contacts.
- Arena: Place where the professional e-sports competitions are performed.
- Party: Central area where people can rest and eat.

- RetroBarcelona: A convention focused on arcade, 8-bit computers and old consoles. A place where nostalgic players can remember and share his childhood, there was also a set of shops selling games hard to find.

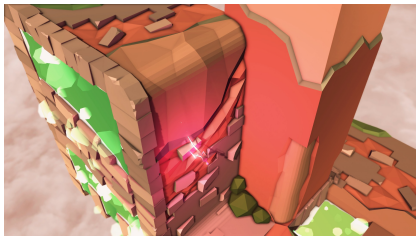


Figure 2.8: Etherborn

Figure 2.9: It's full of sparks

Figure 2.10: Skara: The Blade Remains

When I got to the PC indie zone there were few developers.First of all observed the available games, searching for 3D games with simple graphics and a third person camera.

The first game I visited was *Etherborn*[42], a 3D platformer puzzle game based on gravity alteration created by Altered Matter. The team is composed by Samuel (art lead and game design), Carles (Programming Lead and Game Design), Alexa (Art and Project Management) and Gabriel (Music and Sound FX), all ex-students of the UPC. I first met Alexa, who explained me who they were, their studies (Video Game Design and Development Master's Degree), the game development time (more than a year) and information about the game. Also she recommended me to participate in game jam competitions and the *TIGSource*, a forum to meet game developers and share games. As an artist, she told me the game art creation will be very hard if nobody helps me. After that I talked to Carles, who can explain me the technology beyond the game, a curious fact about the level design is the use of Lego pieces to create real levels to finally put them in the game. As a recommendation and due to my short development time for the project, he said that I must focus on the main aspects of the game and spend more time and efforts to it, instead of trying to do everything perfect.

The next game I visited was *It's full of sparks*[28], a 2.5D puzzle game based on filter colors that modify the environment. I talked to Lluís, the programmer, one of the two developers. The other was Jordi, who created the art and music, with the collaboration of Núria for educational purposes. The game has been developed in one year, overlapped with other projects. The level design is also done with real objects, so it's a recurrent element. One of the most accurate aspects are the colors, special attention to the color palette, and lighting, creating flat cartoon surfaces, finally offering careful graphics.

In both cases I commented that the visual aspect is similar to Monument Valley, a puzzle game about perspective for smartphones with a very unique art direction. They told me that for a lot of developers, this game is a source of inspiration and learning about how lightning and colors are combined.

The rest of the games with available developers didn't fit my concerns, so I bet for a total different game, *Skara: The Blade Remains*[58]. It is a 3D multiplayer free to play based on melee and oriented to e-sports. It is developed by 8-bit Studio, a 40 people company. I spoke with the Marketing Manager. He offered another view about the funding of the company, being some of the investors the team itself and their families. The game is made with Unreal Engine 4, with very polite high-quality graphics and deep gameplay mechanics based on hero abilities. The most important fact was the time between the elaboration of the idea and the development. The director spent 4 years thinking about all the aspects of the game. As a comparison, it took me only 2 weeks to get the initial idea.

After talking with the developers I realized that developing a videogame is mostly a collaborative task among experts in different areas. There is a lot of work that is finally discarded to obtain a valid product and that final users can't appreciate.

2.5.2 Indie videogames monetization

All the non-referenced information in this section belongs to my point of view obtained after several years studying the videogame sector.

Funding

A lot of indie developers find new ways to fund their games without taking an excessive risk:

- Real indie development: The entire budget are self-savings. The risk is very high, but if the game succeeds, all the benefits are for the developer.
- Editor: Big companies like Sony (PlayStation Talents[54]) and Microsoft (ID@Xbox[45]) are becoming interested in the talent of little development teams. The editor provides marketing and promotion in exchange of an important percentage of the sells.
- Steam Greenlight[56]: Steam is the biggest digital PC store, but only companies with an editor can sell on it. Greenlight offers a possibility to indie developers. The system is based on

popularity and once a number of votes are reached, the game is revised and if passes the quality control, it is sold in the Steam shop.

- Crowdfunding: Webpages like Kickstarter[36] and IndieGoGo[34] can be a good starting point to discover if people is interested in your project. You can show concept arts, a technical demo and a description of the idea is enough. The developers decide how much money they need to complete the project, and only if this goal is reached the developer obtains the money.

Marketing

Most of top videogames have a strong an aggressive marketing campaign in TV, Internet and specialized magazines. Indie developers can't compete at this level, principally by the lack of money and contacts. With this handicap they must find a niche market that knows the game exists.

One alternative is offering free copies to analysts and users (e.g. 500 copies in a Twitter's contest). Currently one of the best manners to gain visibility is sending copies to videogame youtubers, people that upload gameplays, analysis, humor.

Launching

Indie games don't require special attention on the launching date like top games, where the peaks sales are on holidays (Christmas, Thanksgiving day). But close launches of other similar games can hide our game. The most important requisite for the launching date is the minimization of bugs, being a playable game. Actually, the $day-1$ *patch* is a common practice to solve or improve the game during the days between sending the last build and the launching date, due to Mechanical Dystopia is a digital product, customers won't have problems to patch it.

Amortization

Like most products, videogames have a Sales Learning Curve. Most of the sells take place during the first week. Since the game is sold at full price, the incomes are very relevant. After the first days, the sales will gradually drop if the price is maintained. The interested people in the game bought the game, or they are waiting for a better pricing. Around the next 6-12 months the price can be dropped

for the patient people. Along the years more price drop can be performed if the benefit prevision is reached to obtain the last marginal sales. Finally, the disruption point of the digital games is the high variability of the price at specific moments, known as flash sales, where basically a lot of non-interested people buy the game due to the discount price. A similar option are the bundles, a pack of games sold with a big discount.

Every company has its own strategies to solve this lack of sales. Companies like Nintendo, Blizzard and Take-Two usually maintain the prices during some years and finally the price is dropped abruptly.

An example of the Sales Curve of a digital indie game[22] is the presented in Figure 2.11. At the game launch a high quantity of sells were performed, with an exponential decrease during the first month. From March to April and from May to June a few sells were made, and only during the flash sales of April and July the number of sells are the half the obtained during the launch day, but with an applied discount.

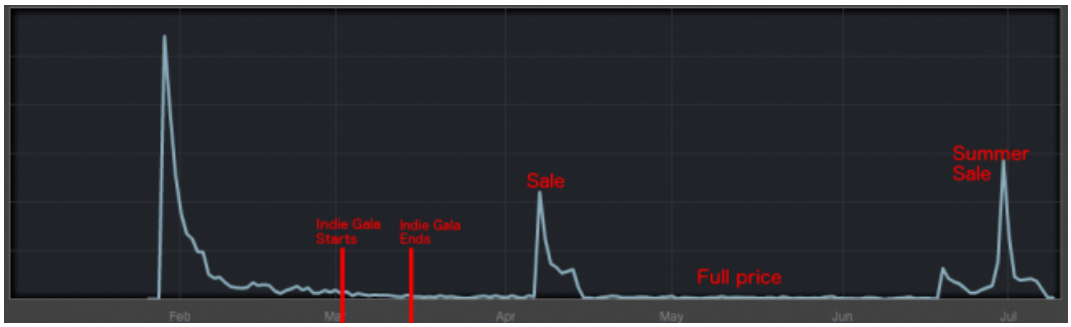


Figure 2.11: Evolution of the sales of a digital game

Death of the product

Once the game has no more chances of being sold, the following alternatives could be taken:

- Sequel: Reusing the core of the game, a new story can be told.
- Downloable Content (DLC): If a new game is too much risky, the addition of new levels, characters or missions could be enough.
- Free updates: Same as DLC, but free. They can be used as a marketing strategy because press will talk about these updates.
- Make it free: In some cases like an imminent sequel or a big DLC, the base game could be free as promotion of other contents.

Chapter 3

Development

3.1 Company

Artifact Gear[6] was born as an indie videogame company with just one employee. Its budget is practically 0, so all the work done by the company is done for free or requiring the permission of the authors for resources like music.

As many indie developers do, a press webpage has been created presenting the company and its games. A very common template is *presskit()*[69], an auto-installable php script that generates all the necessary files to display the pages with a customizable XML file and folders to put resources. The page is hosted on a E2C instance (t2.micro) managed by *Amazon*[7]. The domain was bought at *GoDaddy*[14] (linked using the Route 53 service by Amazon) and the content is managed using *FileZilla*[23].

The URL of the company is: `http://artifactgear.com/press/`. The specific page for Mechanical Dystopia is: `http://artifactgear.com/press/sheet.php?p=mechanical_dystopia`

A link to a shared folder can be found in *Mechanical Dystopia* page to download and test the game.

3.2 Design

3.2.1 Design document

The design document is a template where the designer of a product shapes it, defining all the aspects and the chosen decisions. It details the objectives, functionalities, technology, tools, required roles, temporalization, and its financial aspects. In the case of a videogame, it must firstly define how the game will be, its genre, mechanics, platform target, the controls, plot and art design (including UI, characters, objects and environments). The document must reflect all the thoughts of the designer to be understood by the rest of the team and by founding parties.

Some aspects explained here could be modified during the development of the product, the final features are explained in Section 3.4.

Basic idea

The basic idea is the creation of a roguelite videogame with special emphasis on graphical effects, which impact directly on gameplay.

Development environment

Since I have professional experience with Unity, and I have developed some little projects in my free time, the best option is to implement the game using this engine. have experience in C# language as it is similar to Java, so this will be the chosen language, the other two options (javascript and boo) are less known.

My knowledge level of Unity could be considered as medium, I know its basics and how to use the interface, lifetime of the objects, specific APIs, Physics, animator system, default objects usage, UI system, AI and navigation, particle systems, *AudioMixer*, etc. More or less I have tested all the tools of the engine, but some of them need a deep investigation. So during the project my skills were progressively improved.

The 3D models will be developed in *Blender*. My knowledge about it is very basic, part of the development time will be used to upgrade artistic skills.

The interface, textures and other required 2D graphics will be created using *Photoshop* and *GIMP*. My knowledge of these programs is good.

Target

The game is oriented to roguelite players, that is, people that have already played similar videogames and know their basic mechanics. This market niche can be defined as a minority niche due to it being little widespread principally because of most of roguelite games are indie (normally poor graphics and budget), against other genres like shooters, adventure and sport.

Aesthetics

The aesthetics is characterized by pixelated ingame graphics, being a non-realistic style. There are two approaches, the first is the pixelation of the elements using a shader like *PixelRender* [37]. The other is directly pixelating the whole screen. We have selected the second approach.As a comparative, Binging of Isaac: Rebirth uses a pixel shader for individual elements. We will use a similar level of pixelation but at screen level.

This effect will reduce the accuracy of the models and animations. This type of art provides more flexibility, avoiding disappointment. For example, *Legend of Dungeon*[40] uses voxels with an elaborated illumination system, animations are simple, but the whole impression is still good because is "retro". *MagicaVoxel* [20] allows the design and animation (using *Mixamo* [61]) of models with voxels. It won't be used, but it is an interesting tool to obtain a retro pixelated style.

The use of assets from other people is discarded as long as they don't fit with the rest of elements. Basic elements like generic textures and objects could be used without problems, but most of the 3D objects won't fit since every artist stylizes these in a different way.

Character modeling and objects will be in low-poly 3D using *Blender*. The animations will be created with *Blender* and *Unity*.

Level design

The walkthrough the player must overcome is a set of world divided in levels. After each set of levels, a final boss fight takes place. At each level the user must find one of the exits. They are represented as ground portals.

The scenarios (based on tiles) will contain plain textures with some details and obstacles. In general they will be simple as they will be created randomly.

The view of the player will be isometric or cenital, depending of the observed results once the game is playable. We will take a special caution with walls, enemies and obstacles that may hide the action of the game.

Once the game is playable, the number of worlds and levels by world will be defined.

The final bosses will appear in special designed scenarios fitting their rules.

Story

Roguelite games have usually simple stories, presenting a context and the goal of the main character. A good point of these games is the "hidden lore". The story is told using the objects, enemies, environment and so on, so that the community can join the pieces of this fragmented puzzle, obtaining an approximation to the original plot designed by the developers.

Thematic

The most typical thematics are geographical zones, games like *Nuclear Throne* or *Spelunky*[76] which offer a set of worlds with a predominant environment (desert, ice, city, temple, sea...). Others, i.e. *The Binding of Isaac* or *Legend of Dungeon*, use the same scenario with little variations, normally a dungeon.

In our project, the thematic is important, but not specially relevant for the scenarios. Basic variations of colors or details are enough. For example, Nuclear Throne uses the same level generator for all the levels with different parameter settings on each world. In general, the style of each level is not relevant, only the ice levels have impact in the gameplay.

As a starting point, the thematic of the game will be *steampunk*, but medieval and future age could be considered for other stages of development.

Gameplay

The next sections explains how the basic idea of the game will be performed.

Player

The main character is a kind of magician. He can use magic and melee weapons, one on each hand. The player is able to change the weapons after a level has been completed.

There are different kinds of attack systems: *The Binding of Isaac* uses a constant firerate; *Nuclear Throne* uses ammo, but the most interesting for our project is the stamina, used by *Dark Souls*[47] and *Mass Effect 1*[12]. The character starts without any stamina. Every action adds some stamina. When full of stamina, the character can't perform any action (only move).

There is a unique stamina bar, but in earlier stages of design, a bar for magic and a bar for melee attacks were contemplated.

The stats of the main character are next summarized:

- Health: Points of damage the player can endure before he dies.
- Stamina: Points of effort the player can obtain before he is unable to perform more actions.
- Stamina recovery: Points of effort the player loses at each second.
- Speed: Running velocity of the character.
- Melee attack: Damage multiplier applied to the melee weapon.
- Magic attack: Damage multiplier applied to the magic weapon.
- Melee defense: Damage reduction multiplier applied to melee attacks received from enemies.
- Magic defense: Damage reduction multiplier applied to magic attacks received from enemies.

Effects

The star feature of the game are the effects. They modify the stats of the player, making him more powerful. This is the way the player can "level up", but it also receives some adverse effects in the form of visual nuisances.

Effect	Stat
Pixel	Health
Radial blur	Magic attack
Color	Stamina increase
Chromatic aberration	Stamina
Waves	Melee attack
FOV	Melee defense
Shapes	Magic defense
Shadow	Speed

Table 3.1: Effect-stat relationship

There exist decided three types of effect:

- Permanent: The corresponding bar grows, affecting an effect and stat. It is obtained by picking dropped objects.
- Temporal: Also added to a bar, but its effect decreases along the time until a permanent value is reached.
- Modifier: Some objects can potentiate or palliate the visual effects and stats of the character.By, for example, resetting the effect but maintaining the stat value.

By now, only the permanent effects will be implemented.

There are two possibilities for the effect bars layout (figure 3.1). The first maintains a correlation between the effect and the stat. The second splits them, but an additional bar is required.

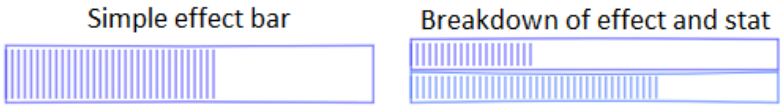


Figure 3.1: Bar effect proposed layouts

During the development, and after an exhaustive search, the 8 effects showed in Figure 3.2.1 were selected and attached with a stat.

The rest of investigated effects are: ASCII, chromatic aberration on elements, chromatic aberration with circular movements, broken screen, blur specific elements, altering the drawing system (black and white, Chinese ink, high color saturation, cartoon style).

The modification of the drawing system was one of the motivations to develop this project.

Keyboard/Mouse	Xbox 360	Action
WASD	Left Stick	Move the character
Mouse aim	Right Stick	Direction of the character
Left click	LT	Left weapon attack
Right click	RT	Right weapon attack
Space	RB	Dash
Esc	Start	Pause

Table 3.2: Ingame controls

Weapons

There are two kinds of weapon: melee and magic weapons. Each weapon can only be used in one hand: melee on the right, and magic on the left.

The melee weapons are weapons existing on the XIX century, that are: swords, hammers, spears, axes, and so on. Some other objects that aren't weapons have also been introduced: a guitar, a pen, an umbrella...

Magic weapons are related to the left mechanic arm of the main character, so in general are considered magic spells, but in the context of the game, they are science. The spells are: fireballs, flamethrowers, lightnings, wind gusts, different kind of poisons. Like the melee weapons, the look and feel can be adapted easily. The important part is the behavior.

Every weapon has an assigned melee or magic damage and a specific behavior, including a unique animation.

Objects

Once an enemy is killed, it drops an item related to an stat and effect. If the user gets it, the desired stat/effect is increased. This increment will be tuned once the game is completely playable.

Controls

The game can be played with mouse and keyboard, or gamepad, specifically the well-standardized *Xbox 360* controller. Unity has a good navigation UI system for gamepads, but by now, only the ingame aspect will be available for gamepad due to mouse input UI is faster and easier to set up.

The ingame actions the player can perform are presented in Figure 3.2.1

Enemies

The enemies the player must face are robots. Every robot has a specific behavior. As a proposal:

- No movement and shoots periodically after few seconds
- Pursues the player faster and perform melee attacks
- Sniper
- Random walker
- Random jumper
- Walks slowly to the player and shots
- Shots if it is far from X distance, else, perform melee attacks
- Bomb thrower
- Shots lot of projectiles in a short period of time
- Healer, including the player (but very few), but also can attack
- Apply temporal effects to the screen or player

As a simplification, the same enemy can have different colors, splitting it in difficulty tiers. The more the user advances the more hard to him the enemies will be.

Decisions

The main dilemma of the player is which effects he prefers to choose to progress in the game. Also, at the end of the game the player must choose between two options out of a total of three possible endings.

Difficulty

Roguelite videogame are usually a hard challenge for players. In this case, the game will be difficulty by itself due to the effects feature, but also the enemies will be hard to beat.

Continuity

Games like *Crypt of the Necrodancer*[26] and *Rogue Legacy*[27] promote the collection of objects to improve the stats of the following character, making the game easier every time and reaching further levels. Others unlock new characters and objects, like *The Binding of Isaac* and *Nuclear Throne*. *TBoI* also continues once the end is reached, being able to obtain new endings. Finally, others don't save anything, the first game is equal to the 100th. In the current project, only stats about the runs will be stored.

Music and SFX

The music must be chosen according to the retro style. We have chosen chiptune and 8-bit genres, being synthetic songs that every computer of 8/16-bit console can emulate. We will be used existent songs, after checking their licenses.

Unity provides methods to analyze the spectrum of audio, so it is possible to move objects according it.

GUI

An initial design of the ingame GUI is showed in the figure 3.2. It includes the dual stamina bars, so that it is outdated. The GUI must be simple but display all the necessary data for the player. The simplest GUI is the classical health and stamina bars at the left-top, and the effect bars at the bottom. It can be improved by experimenting with different art styles, positions and sizes. The expectations are the use of elaborated shaped buttons and bars.

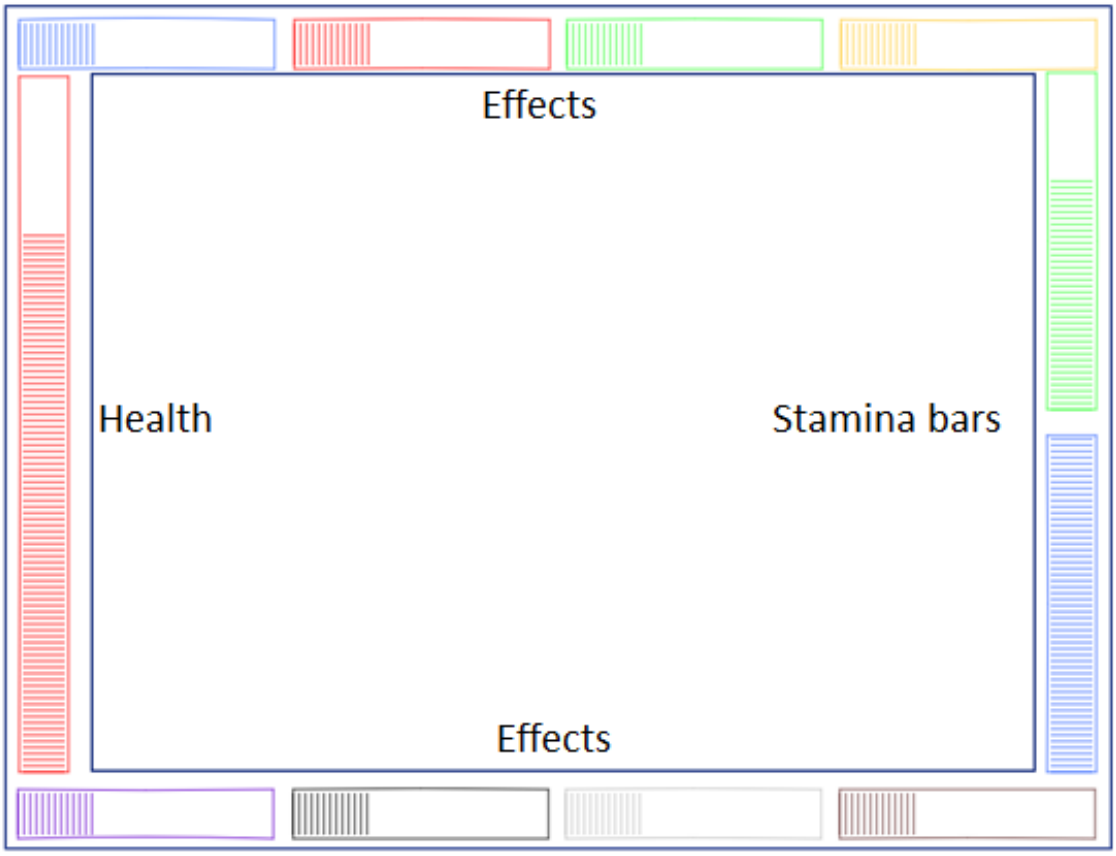


Figure 3.2: Initial ingame GUI

Targets

Unity allows to export your project to most the current platforms. If all the assets and code are compatible, only changing the current target, a new build can be done for it. It is usual for a medium/big project to find incompatibility of some parts, and performance issues.

The base platform is Windows, with an easy exportation to Linux and Mac. Smartphones could be an interesting market, but the difficulty of the game would increase with touch controls, so a physical gamepad would be required. Some tests can be done for Android, iOs and Windows Phone, but by now, only PC is contemplated.

Unity also supports the web player target, working as a "Flash navigator game".The deployments for testing could be done using this target as long as no incompatibilities appear.

Finally, there are easy plugins to convert a normal project to a VR project by simply replacing a

single camera with a dual-eye cameras.

Monetization

The best way to obtain a minimum ROI and media repercussion is trying to publish the game using the *Steam Greenlight*[56] platform. The first builds can be completely free as you needed testers to obtain feedback and detect bugs. Once the game enters on alpha version, that is, *Steam Early Access*[55], the proposed price is a symbolic euro. Now people will contribute requesting features and criticizing how the development is going on. The final build at the Steam shop should take a price around the 5 euros, with 30% of gain for Steam.

3.2.2 Source of inspiration

The greatest exponent of the roguelite videogames is *The Binding of Isaac*. All the games of this genre get inspiration from it. Specifically, the procedural level design focused on rooms with a final boss after the exit is found is the main contribution. A common genre related to roguelites is bullet hell, lots of projectiles on screen the player must dodge. One of the most valued is *Nuclear Throne*. Finally, the mixture of these to games originated *Enter the Gungeon*.

Other games that are not roguelite but have common features are *Hyper Light Drifter*[41], *Bastion*[60] and *Stories: The Path of Destinies*[31]. They are adventure games with very similar gameplay including melee and ranged weapons, dash and a cenital/top-down camera.

The thematic of *Mechanical Dystopia* is steampunk. Initially the game was raised as an age evolution, like *Crash Bandicoot 3*[2] (ancient Egypt, ancient China, medieval, World War I, future).

Steampunk[48] is a literary movement originated at the end of the 80's as an evolution of cyberpunk (based on a near high-tech future). This genre is based on the British Victorian era, most specifically on the industrial revolution, with a high prevalence on gears, steam, leather and metal, producing clothes, gadgets, ornaments and scenes heavily overloaded with a predominance on brown and ocher tones.

3.2.3 Context

Year 1875. Engineers have been using steam and carbon during the last 25 years as an energy source. Lots of new inventions were introduced to society: vehicles, medical devices, heating in houses, weapons. The trending technology is the mechanized assistant, an autonomous device that helps people, with a basic artificial intelligence to perform assigned tasks. These machines removed the bindings of housework, repetitive tasks and most of the jobs have practically disappeared. As a consequence of the activity of the robots, toxic byproducts are generated. People started to experiment with these substances, called essences, and discovered some health benefits, but also a disruption of the senses...

3.2.4 Plot

Some sections are marked as spoilers, so if you don't want to know the end of the game or other works, avoid them.

Approach

Jasper Herrington, a retired military commandant, was working in his workshop repairing defective robots when he noticed that the floor was trembling and all the robots were fired at the ceiling, breaking it. There seems to be a strange force attracting all metallic objects to the sky, including Jasper.

Jasper is not a usual mechanic, during the Steam War he lost his left arm, making him unable to maintain his military position. But he has big hopes to recover his arm with the new prosthesis powered by steam.

After some years of retirement, Jasper was gifted with a utility arm by the Government. Some time later he was able to modify his arm by replacing parts of it (specially the hand) and becoming able to manage tools with it.

Due to the rise of mechanical robots, Jasper set up a construction and repair workshop for a wide range of models, being able to repair them using his multipurpose arm.

Knot

Jasper appears in a dungeon full of aggressive robots. The unique option is to clear rooms of robots and reach the portal to the next dungeon. After each world, a boss fights with Jasper, trying to convince him to surrender.

Outcome (spoilers)

Jasper finally reaches *The Creation*, the final boss of the game. Once the fight starts, the boss raises a dilemma about its motivations. The player must choose one of the two options:

- Agree: The game ends without fighting.
- Disagree: As other boss stages, *The Creation* attacks the player who must defeat it.

This decision flows into three endings:

- Agree: Robots will live apart from humans, who must live without the help of the machines, starting a pseudo-medieval age.
- Disagree and player wins: *The Creator* is destroyed and its influence doesn't affect any more to the other robots. Humans will continue its own *Mechanical Dystopia* with the benefits and losses that this entails.
- Disagree and player loses: The Creator orders a robot invasion to defeat the humans.

Moral (spoilers)

In the game story, not everything is good or bad.The final decision is chosen by the player, being responsible for the consequences.

As many other games do, not everything will be explained to the player. The community who complete the story gaps. The next information can be one of these hidden lore:

Looking for ways to cause planned obsolescence to the new devices that were coming, Jasper created a new material capable of generating awareness combining it with metal, being the steam the main actuation engine. The experiment was a success, causing the devices to suffer unexpected accidents, but in a reasonable way to think that they have been provoked by external causes. Once a device is

broken, someone like Jasper must repair it. But the fact of endowing intelligence, however minimal, implies "free-will". Some of the machines ended up learning to collaborate with others and erased the idea of obsolescence, transcending to a new type of living being, and perpetuating a plan to finish with its creator.

Inspiration (spoilers)

First of all, the idea that Jasper has a mechanical arm is not new. Other characters like Adam Jensen from *Deus Ex*[18] (both arms are metallic), Edward from *Fullmetal Alchemist* or Big Boss in *Metal Gear Solid V*[38] haven't it, providing new abilities with this pretext. Other games offer a left powered hand, and a weapon in the right hand, like *Bioshock*[1] and *Dishonored*[59].

The robot rebellion is heavily exploited in the literature (I, Robot), cinema (Terminator) and videogames (Binary Domain[51], Mass Effect). In this case, some of the robots have consciousness, which are managed by a bigger entity that has the power.

The essences are basically *soma* from *Brave New World*, *ADAM* from *Bioshock* and *red sand* from *Mass Effect*. The integration with the gameplay is well solved. Instead of being a consumable object that enemies drop without any reason.

3.3 Prototype

Once the project design defined the basic concepts of the game, a little prototype was developed to provide a better vision of the concept. The proposed milestone for two weeks of work was the next:

- 2 effects
- Player character
  - Movement
  - Dash
  - Melee and weapon attacks
- Player character can dash
- Statistics management
- Map

- An enemy
- A final boss
- Collisions among characters, melee and weapon attacks and scenario walls
- Main menu
- Pause menu

In the prototype the player can start the proposed level from the main menu. There is just one type of enemy, a capsule that goes to and shots the player using *RAIN AI*. The weapons of the player are a sword and firebolt magic. The main character can also dash. The effects implemented were pixel and blur, and a third effect created two years ago by me, the color effect. The map was partially integrated an at this stage some physics and scale issues occurred. The GUI was also implemented, the health and stamina bar, and the effect bars, all them linked on the stats of the player, upgradeable using debug functions.

As a starting point, the prototype helped to decide some aspects like the camera orientation, player velocity, effects evolution, notice which parts are more difficult to implement, and what useful assets are available on the Unity Store.

Finally, the AI final boss was not implemented due to the complexity of *RAIN AI* but the character class hierarchy about characters was completed. Other functionalities were implemented to supply the lack of the boss AI. There were the trail of the melee weapon, the GIF recorder, the skeleton of the save system and a basic game manager (only including player's death event and the game over menu).

3.4 Game Overview

In this section we present all the content available in the game, the actions the player can do, how to play, how the enemies and final bosses are, which weapons and items are available, how to craft new objects, and the game mechanics the player could check to play better.

As part of the roguelite mechanics, the player should not know the relation between effects and stats, their evolution, the use of essences, which the best weapons are, how the strategies to defeat the enemies are and all the information explained here.

3.4.1 Main menu

The game starts with splash screens showing the *Unity* logo, the company (*Artifact Gear*), Universitat de Lleida and the creator of the game.

Figure 3.3: Main menu

In the main menu (Figure 3.3) there are five displayed options:

- Play: Player starts the game generating a random run if he doesn't select any random seed or performing a specific level if he does. The first time the game is started, the story button is displayed and then the game starts. After that, the story button becomes available at the top left of the window. Also, if player saved a game you can continue it.
- Replays: All the failed runs will be stored here, displaying information about them, a button to see how player died and another one to play again a specific run.
- Options: The player can change general music and effects volume, decide which camera style he prefers and the language. The available languages are: English, Spanish, Catalan, French, Italian and Romanian.
- Credits: The credits of the game are displayed, including who participated on it and the assets and music used.
- Exit: Simply to exit the game.

3.4.2 Main Character and controls

The main character, Jasper Herrington, is able to move around the rooms in a level. It can do a dash to avoid enemy attacks and projectiles, and attack with a melee weapon and magic. Every time Jasper does an action, its body overloads. If the stamina bar (blue bar) becomes full, he can not do more actions during two seconds. There are two more actions related to the game, pause and visual help. The visual help displays Jasper and objects without graphical effects to avoid color recognition problems or see hidden elements. If the health of Jasper (green bar) falls to 0, he dies and the game over screen appears.

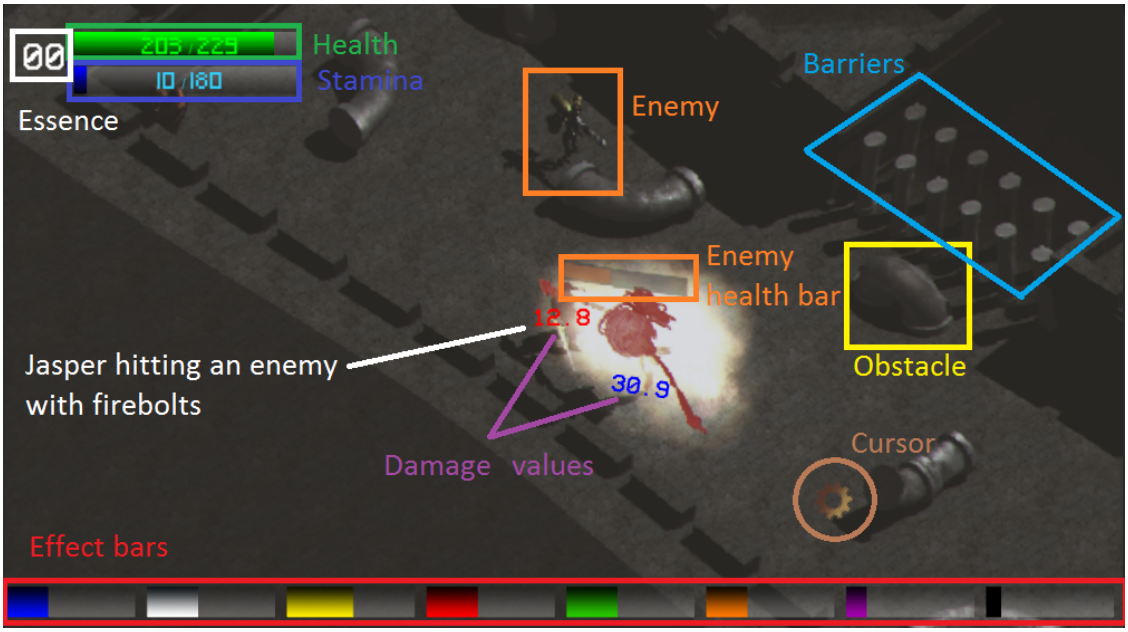


Figure 3.4: Ingame screenshot with explanations

Jasper can be controlled with the keyboard and the mouse, or with a gamepad like *Xbox 360* and *PS4* controllers.

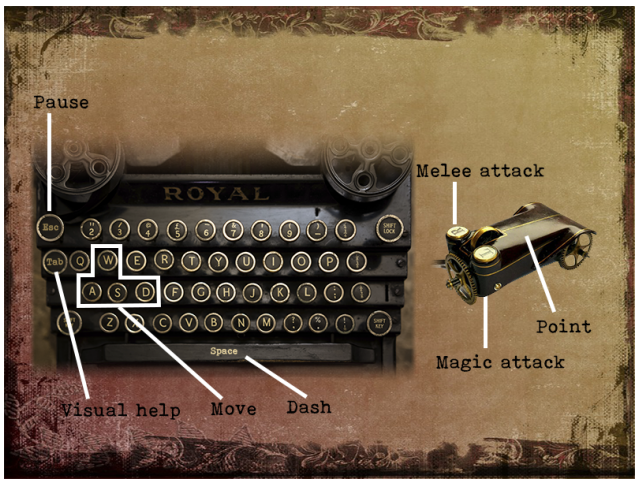


Figure 3.5: Keyboad and mouse controls

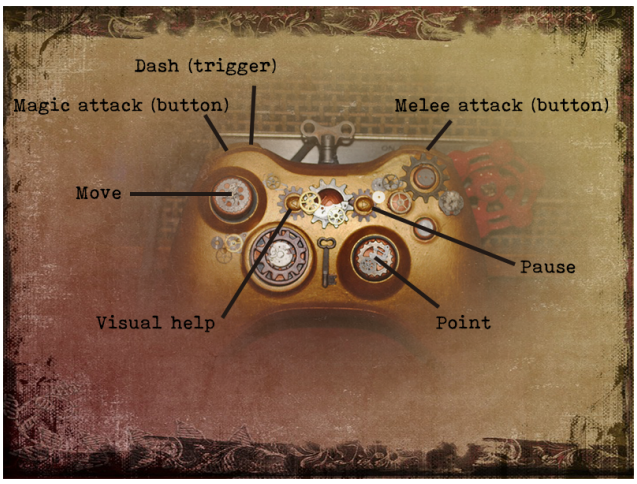


Figure 3.6: Gamepad controls

Finally, there are two available perspectives:

- Orthographic: This is the default and recommended perspective (Figure 3.7). The camera is over Jasper, the player sees all the area around Jasper and it is easy to attack in all direction. The detail level is low.
- Behind the character: The camera is attached to the player (Figure 3.8). The player only sees the area in front of Jasper and it is more difficult to attack. The immersion and graphics are higher.

The perspective can be changed from the options inside the main menu.

Figure 3.7: Orthographic view

Figure 3.8: Behind character view

3.4.3 Effects

At the bottom of the screen you can find the effect bars. The player must get objects, called essences, dropped by the enemies to increase his statistics. Each essence type is related to an effect bar. The

fuller the bar, the higher the statistic benefit and the graphical effect. The statistic benefit scales linearly with the effect bar, but the graphical effect only starts after the completion of 30% of the bar.

Pixel effect

By default, the screen is pixelated to obtain a retro style. The increment of this effect makes the resolution of the game decreases. At the end you obtain enormous pixels. The health stat is improved.

Figure 3.9: Pixel effect off

Figure 3.10: Pixel effect on

Radial blur effect

The screen becomes blurry depending of the position of the cursor. The parts close to the cursor keep clear graphics, while further zones get fuzzy. The magic damage is improved.

Figure 3.11: Radial blur effect off

Figure 3.12: Radial blur effect on

Color effect

This effect increases the stamina cooldown but it provokes the modification of the color palette and the contrast.

Figure 3.13: Color effect off

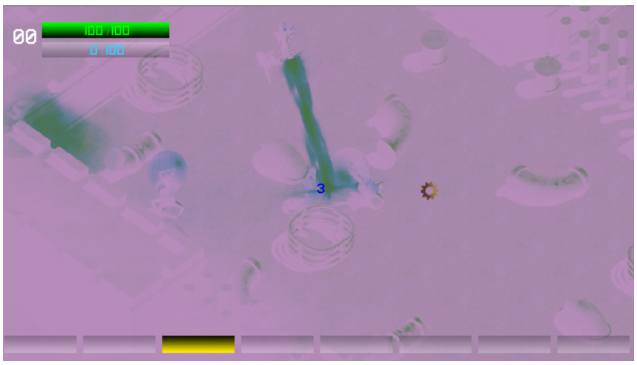


Figure 3.14: Color effect on

FOV effect

It increase the melee defense but the borders of the screen becomes curved.

Figure 3.15: FOV effect off

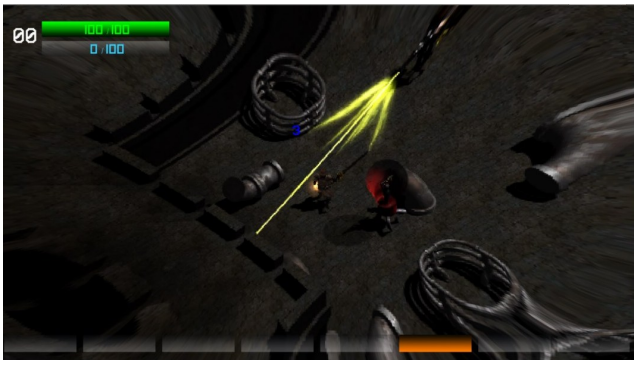


Figure 3.16: FOV effect on

Chromatic aberration effect

The RGB (red, green and blue) color channels are split very fast, performing graphical vibrations. The benefit is an increase of stamina.

Figure 3.17: Chromatic aberration effect off

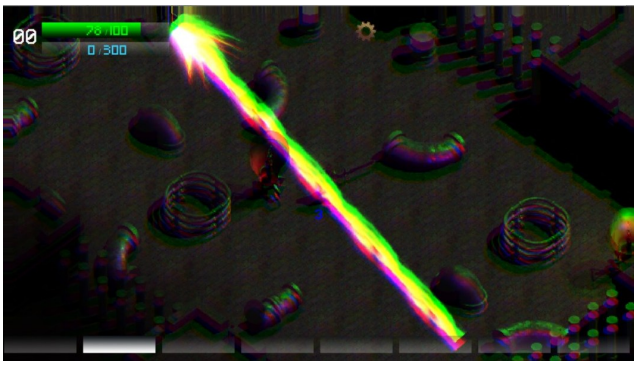


Figure 3.18: Chromatic aberration effect on

Alter shape effect

A group of colored squares crosses the screen from one side to another, increasing the magic defense of Jasper.

Figure 3.19: Alter shape effect off

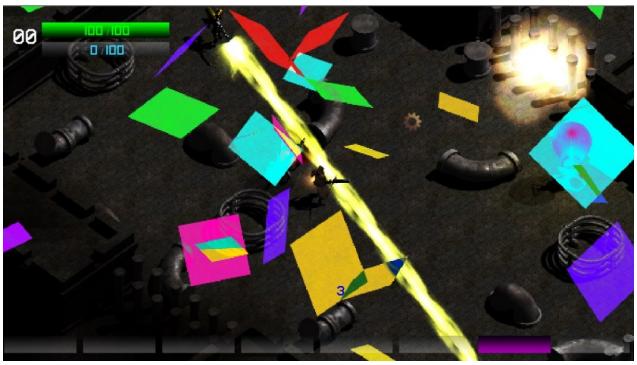


Figure 3.20: Alter shape effect on

Waves effect

Melee attack is upgraded, but the screen curls like waves.

Figure 3.21: Waves effect off

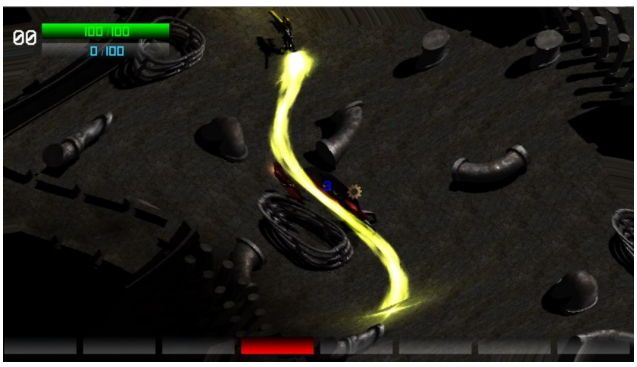


Figure 3.22: Waves effect on

Shadow effect

The speed of Jasper increases, but the environmental light reduces. Luckily, Jasper has its own light, so the area near him is lit.

Figure 3.23: Shadow effect off

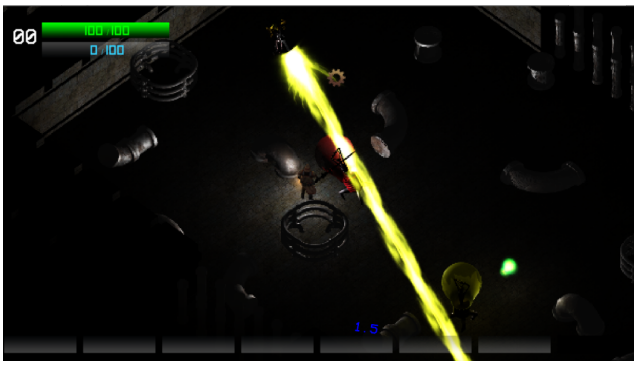


Figure 3.24: Shadow effect on

3.4.4 Levels

The game is composed of worlds and levels following the next schema:

World	World 1				World 2			World 3				World 4				
Level	1-1	1-2	1-3	Training Train	2-1	2-2	Lightning	3-1	3-2	3-3	mAlice	4-1	4-2	4-3	4-4	The Creation
N# level	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tier	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	5

Table 3.3: World route

The player must complete 16 levels to finish the game. Once the last dungeon level of each world is reached, you engage in a fight with the final boss of the world.

To finish a level, Jasper must enter a portal. Portals appear after the completion of given number of rooms. Depending on the number of rooms in a level there can be more or less levels. Table 3.4.4 shows when they appear. Initially every effect has the same probability to appear, but entering a portal alters these probabilities, increasing the probability of the essence type with the color of the portal.

The completion of levels allows the increase of the item tiers, the more levels completed, the most probability to obtain better items.

Table 3.4: Level size and appearance of portals

		% completed rooms		
Level type	Level size	First portal	Second portal	Third portal
Small	n# rooms <= 10	70% - 100%	Last room	No
Medium	10 <n# rooms <= 20	60% - 80%	% of first portal - 100%	Last room
Big	20 <n# rooms	60% - 70%	70% - 80%	Last room

3.4.5 Enemies

Every world has a set of enemies that can appear on it with a given probability. When player enters an unvisited room, the enemies are generated at a reasonable distance considering the position of the player to avoid unfair situations. Every enemy has a set of routines, and it attacks the player who has to to avoid them. Also, not all enemies can be killed with the same weapons. Like the character has melee and magic defense, enemies also have. They can vary depending of the state of the enemy. After an enemy is killed, an essence is dropped, the player has ten seconds to get it before it disappears. The player can only exit a room if all the enemies have been defeated.

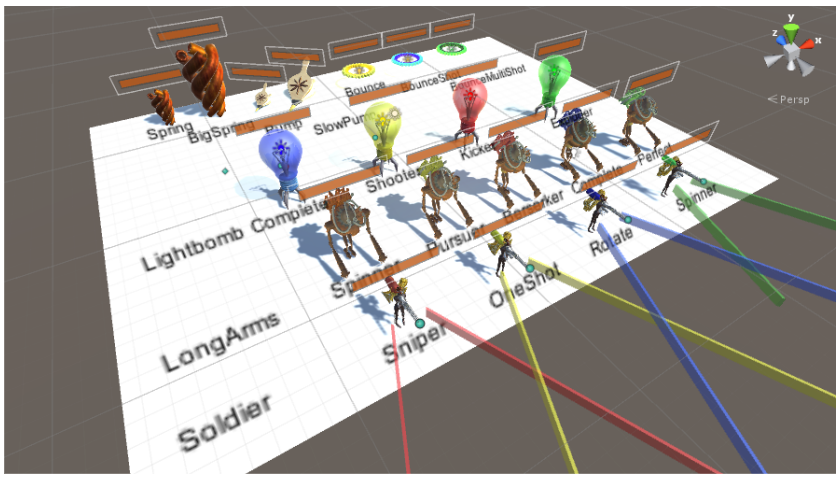


Figure 3.25: Enemies

The next sections explain the behavior of each enemy and its lore:

Spring family

Cables and tubes, everybody needs them, for water, steam, devices... But nobody cares about them. They are always in the floor, dirty, wet, peeled, leaky. As a king of rats, some cables become a new entity called spring, with collective consciousness and eager to destroy their masters.

A spring follows the player when he enters in its visual range or if it gets damaged. If it is closer enough to the player, it does a melee attack.

Types

- Spring: It has the default behavior of the spring family.
- Spring Slow: It is a bigger and slower version of the default spring.

Pump family

Pumps are used for heating, cooking and in industrial processes. They are proud to be steam and fire generators, but they finally joined the cause of the rest of the machines.

A pump starts to follow the player if he enters into its visual range. They launch fire, provoking a durable damage.

Types

- Pump: The default behavior of a pump implementation.
- Pump Big: It works exactly like the default pump, but it is bigger, slower and causes more damage.

Bounce family

Most of the gears spin without a rest. Its unique purpose is to be part of a bigger machine, who obtains all the merits and attention. Now gears want their own recognition acting independently.

Gears patrol aimlessly, bouncing on walls and enemies. The contact with them provokes damage.

Types

- Bounce: The default implementation of bounce behavior.
- Bounce Shot: When a bounce shot is moving, it damages by contact. When it is quiet it doesn't. So the best strategy to defeat it is waiting until it shots an energy ball, dash near it, and do a melee attack.
- Bounce Multi Shot: It has the same behavior as Bounce Shot, but it launches a variable number of energy balls.

Lightbomb family

Lampposts are the progenitors of the lightbombs. At night everyone can have a light thanks to their mobility. Since electricity is currently unstable, steam has been imposed to this energy, being one of the few devices with this technology. Lightbombs are used as explosives when riots occur.

Lightbombs have a variety of behaviors like walk, melee attack, shot and explode.

Types

- Lightbomb Kicker: Kicker goes to the player and perform attacks with its head.
- Lightbomb Shooter: Shooter moves randomly and shots in the direction it is facing to.

- Lightbomb Exploder: Exloder goes slowly to the player, shooting directly against its position. If its health is 50% or lower, it performs an explosion. Once the explosion occurred, it can only attack with head attacks as kicker.
- Lightbomb Complete: It works as the first phase of exploder. It goes to the player but it doesn't explode.

Long arms family

Long arms is the friendly servant for the entire family. Its arms are able to do lot of craft works as paint a wall, clean dishes, and cooking. Also, it can be a dog walker, a babysitter and a guardian. So this kind of robot is one of the most exploited.

Long arms have different attacks based on pursuing the player with melee attacks. They have the capacity to find the shortest path to catch the player.

Types

- Long Arms Berserker: When the player is in front of a berserker, it starts a fast run in a straight line, trying to collide against the player.
- Long Arms Pursuer: Pursuer works like a berserker, but it has the capacity to follow the player during the pursuing.
- Long Arms Spinner: When the player is near a spinner, it performs its spin attack following the player.
- Long Arms Complete: A complete is a combination of a behaviors of berserker and a spinner.
- Long Arms Perfect: A perfect has the attacks of a pursuer and a spinner.

Soldier family

Working in the border posts, sentinels are failed cyborg soldiers. They were directly controlled by their former superiors. In order to avoid removing the neural implants, their arms were sectioned, so its unique task is looking out and shooting when required.

A soldier maintains its position but it is able to face the main character. The implemented version is the sniper type. In all its version it shoots a laser. They are able to detect the position of the player,

but each type has a different behavior.

Initially Soldier was the class to implement different kind of soldiers, one of them Sniper. Due to the good results (but time costly) of the laser implementation for Sniper, using the asset Basic Beam Shot, the other projectiles like grenades, bullets, missiles and so on were canceled.

Types

- Soldier Sniper: The default version shots a low-damage continuous laser.
- Soldier Sniper One Shot: One Shot, as it name suggests, does a quick high-damage laser shot.
- Soldier Sniper Rotate: Rotate is an improved version of the default sniper, it shots a continuous laser but it can also rotate, following the player.
- Soldier Sniper Rotate Spinner: Rotate Spinner starts to spin when the player enters its visual range. When it turns, the continuous laser curves.

3.4.6 Final bosses

At the end of each world, Jasper must defeat a stronger enemy to advance to the next world. A final boss is a powerful enemy normally found at the end of a level or world. Most games show a health bar directly in the HUD. The behavior of the boss is a chain of attacks with some delays in between. Once it is defeated an artifact is dropped. In this case, these basic rules are followed, so users can understand easily how a final boss level works. The game has four bosses.

Training Train

This train was managed by the newbie machinists during their practices. Nobody takes it into account it when the steam is not enough. It was usually in maintenance in the Jasper's workshop. Jasper always fixed it, extending its torment instead of removing it definitively.

Actions

- External travel: It wanders the stage outside throwing fireballs.
- Fire tornado: In the center of the screen, it spins to throw a huge quantity of fireballs.

- Break: It stays cooling during some seconds in the center of the scenario. This action doesn't damage Jasper.

Lightning Mask

Created under unknown circumstances during a thunderstorm inside the old Steam and New Energies Research Center, the mask is similar to those carried by scientists. Maybe it is one of them. Who knows... And the chains? Could it be a prisoner?

Actions

- Central Attack: The towers move up and start to charge lightnings at the center. After a short delay, a big explosion affecting around the 90% of the scenario occurs. The player must be in the bounds to avoid the effects of this attack.
- Patrol: The boss selects a point of the map and moves to it at constant velocity. Once reached, it selects another.
- Shooting: *Lightning Mask* shoots a set of waves of electric balls directly against the player but with a big dispersion and a different velocity for each projectile. The less health, the more fire rate is executed.
- Towers Attack: Two towers are connected by a lightning. Combined with Towers Rotation action, the player must evade it running and dashing.
- Toroid Attack: *Lightning Mask* goes to the center of the scenario and launches lot of lightnings. The player must be close it to avoid the attack. Also, the player can take advantage and hit it without risks.
- Swap Towers Rotation: The direction of rotation of the towers is changed.
- Towers Rotation: The less live the boss has, the more angular velocity the towers obtain.
- Wait: When the battle starts, there is a little delay to prepare the player.
- Death Attack: When *Lightning Mask* dies, an attack similar to *Towers Attack* is performed, but in the current position of the enemy.

mAlice

Once steam became an usual power supply, paradoxically, one of the first jobs that disappeared was that of firefighters and rescue teams. People called them *Angels of salvation*. Fire remains a problem for their bodies but they can appease it with their wings and put people in a safe place. The halo is more a symbol than a useful complement.

Actions

- Patrol: It moves to selected points.
- Left hand shooting: A set of projectiles are thrown to the player.
- Right hand shooting: Another set of projectiles with more dispersion is shot.
- Spin pursuer: *mAlice* turns on itself trying to reach the player.
- Lasers: Two lasers appear on their wings.
- Wait: Between action and action there is a little break.

The Creation

The ultimate weapon of the machine revolution. It seems to be a human creation. Its advanced circuitry, powerful projectiles and attentive perception of the world are exceptional. Who has the capacity to assemble that perfect entity? Someone like... Jasper?

The interface for this boss is different. The playable screen becomes smaller than the rest of the game. At the top right there is the sight of The Creation and the rest of screen simulates the game is executed on the Unity editor.

Actions

- External lasers: The external ring of the scenario throws six lasers focused on the center and it rotates.
- Internal lasers: Same attack as external lasers, but they appear from The Creation position.
- Combined lasers: The two mentioned groups of attacking lasers executed at the same time.
- Pursuing missiles: Several missiles appear into scenario and pursue the player. After some time

they explode automatically.

- Shooting: A large number of bullets are fired in multiple directions.

3.4.7 Weapons

There are two types of weapons. The player carries one weapon on each hand. He must uses them to defeat enemies. The player can know if an enemy gets damaged because the enemy gets a red tonality during a short period of time. The health bar over it decreases and a number (the damage done) is shown close to it.

Melee

Melee weapons have a short range that depends on their length and shape. There are short weapons like*Basic axe*, medium range like *Clous sword*, the hammers like *Steam hammer* only do damage with the mallet. Others like *Reforged blade* comprise large arcs, and others have several hits like *Gear axe*. There are 13 melee weapons.

Figure 3.26: From left to right, Basix axe, Gear axe, Cloud sword, Steam hammer and Reforged blade

Magic

Jasper can also cast magic spells to kill enemies from a greater distance. There are two main types of spell: throwable balls that cause damage when they impact on enemies; and continuous damage spells, that cause damage by contact along the time. There are 13 magic spells.

Figure 3.27: Firebolt magic spell

Figure 3.28: Flamethrower magic spell

3.4.8 Workshop

Once Jasper enters a portal, he has some time to prepare for to the next level. In the workshop, the player can spend the essence obtained from the enemies to obtain items, melee and magic weapons. Every object (including weapons) comes with an upgrade of the statistics and a reduction of the graphical effects (but this also implies a reduction of the statistics). The player must search a balance between the number of essences, statistics and graphical effects issues to be strong enough to defeat the enemies without experimenting excessive visual problems. Every item has a tier. In the workshop only items with the same or less tier than the available do appear.

Figure 3.29: Workshop

3.4.9 Items

When a final boss is defeated, it drops an artifact (box with text in Figure 3.30) to upgrade the statistics of Jasper. Unlike crafted items in the workshop, the ingame items don't reduce the values

of the effect bars. There are 100 artifacts. Like the workshop items, they have specific tiers, so that the most advanced the run, the more artifacts are available.

Figure 3.30: Artifact and health items

There are also items to increase health (green crosses in Figure 3.30). The more damaged Jasper is, the higher the probability to obtain a health item. They can be small, medium or big, recovering 10%, 25% and 50% of the health respectively.

3.4.10 Game mechanics

Melee attacks, magic attacks and dash overheat Jasper. The player can not abuse of their actions since when stamina gets is full, there is an important time penalty until stamina decreases again. The player should act prudently to be able to escape if the situation gets hard.

Every level is composed of rooms. After beating a determined number of rooms, a portal in the center of the last room completed will appear. The player must decide if the color of the portal is interesting for him.

The rooms are full of obstacles, the player can use them to avoid enemies and projectiles.

In the workshop, the player can convert essences into items and weapons, reducing the impact of the graphical effects, but also their benefits in the statistics. The player can decide to spend essence to reduce the graphical effects and gain an upgrade in their statistics or wait for the next level to obtain better objects at the cost of having to endure the graphic effects.

Enemies have melee and magic defense, but these can vary depending on the state of the enemy. The player must discover their strengths and weaknesses to defeat them more easily.

3.4.11 Remarkable features

The game can be saved in the workshop so as to continue the run in another moment. Once the player continues, the saved slot is deleted to avoid traps, e.g. repeat a bad run, try to obtain a better object in the workshop, or amend player failures.

When the player dies, the game over screen shows how the player was defeated.

In the play menu, the player can decide to start a random run, or put a combination of 10 numbers (called seed) to start a specific run.

The final bosses and their scenarios have decorative elements that are affected by the rhythm of the music. Also, when a boss battle starts, a floating camera shows the scenario and the boss.

The story of the game is not presented directly. There is an initial plot at the start, but the rest is not shown. It can be found distributed in this report. Also, most of the mechanics are explained here, but in a real situation, the player must discover how to defeat the enemies, which are the most effective weapons and the best strategies.

The game has a huge number of statistics for Jasper, enemies, melee and magic weapons, items and graphical effects. These statistics should be balanced by a group of testers, statistical and QA (Quality Assurance) to tune them better. All the content except the statistics upgrades are handmade, searching a logical and a fair progress. The statistics upgrades are generated randomly following bounded formulae and rules.

3.4.12 Debug options

In order to check the contents of the game, the player is able to execute cheats. Logically a normal game does not make these cheats available, but without them, most of the content probably will be unreachable or unlikely to be found by now.

The cheats for the main menu are displayed in Table 3.6, and the cheats in-game are in Table 3.8.

Key	Result
1, 2, 3, 4	Load the corresponding world
5, 6, 7, 8	Load the corresponding boss (Training Train, Lightning Mask, mAlice, The Creation, respectively)

Table 3.6: Main menu cheats

Key	Result
C	Invincible, no stamina cost
X	Kill all enemies
Z	A portal appears in the current run (not available in the first room)
Right control + numeric keypad (numbers, +, -, *)	Change the melee weapon
Numeric keypad (numbers, +, -, *)	Change the magic weapon
Numbers from 1 to 8	Gain power to the respectively effect bar
Left Alt + numbers from 1 to 8	Subtract power to the respectively effect bar
0 (zero)	Set all the effect bars to 0
J	Gain a random essence
U	Spawn a random artifact
I	Spawn a health item
H	Obtain 10 essences
N	Slow the time
M	Accelerate the time
F	Show/hide FPS counter

Table 3.8: In-game cheats

3.5 Code

The following section explains how several functionalities have been implemented: the structure of the project, how the graphical effects are integrated, the implementation of the combat system, the hierarchy of characters, the global game logic and how the UI was created.

Most of the subsections are independent and can be understood by themselves, but others require a global vision of the project or a prior reading of some related subsections.

All the code documentation generated automatically with *Doxygen*[17] can be checked at:

`http://artifactgear.com/documentation/`

3.5.1 Structure

Project structure

Due to the elevated size of the project, all the content must be classified correctly so that it can be found quickly. Figure 3.31 shows the main folders hierarchy. Inside each folder there are additional subfolders like menus, ingame, enemies, environment and textures.

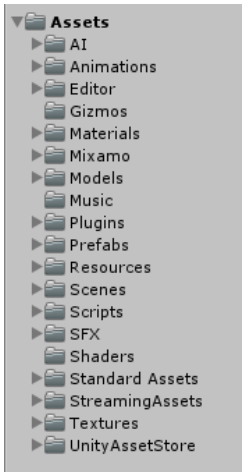


Figure 3.31: Main menu

Game structure

As any other *Unity* game, *Mechanical Dystopia* is composed of scenes. Figure 3.32 shows the scenes of the project and the relations among them.

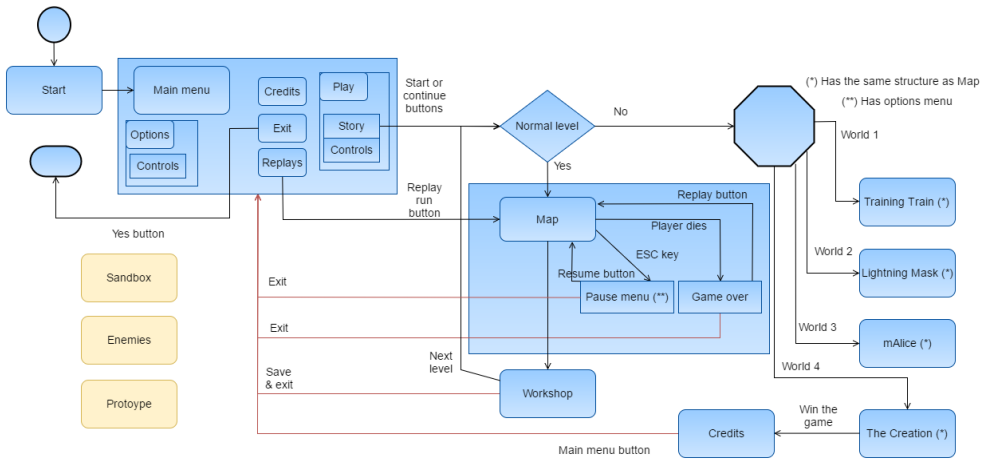


Figure 3.32: Workflow

- Start: A void scene to initialize data.
- Main menu: The first scene the player sees and where he mainly can start or continue a run.
- Map: All the levels of the game are generated using this scene.
- Workshop: The menu where the player can spend essence to update his statistics after he completes a level or kills a final boss.
- Training Train, Lightning Boss, mAlice and The Creation: The final bosses have their own scenario since each one requires a handmade implementation and configuration.

- Credits: A specific version of the credits presented in the menu, it is displayed when the player finishes the game.

There are scenes unreachable to the player, employed for debugging and testing:

- Prototype: The first scene created in the project to implement the game prototype.
- Sandbox: It is the main environment to test the behavior of enemies.
- Enemies: It is used to configure the statistics of each enemy more easily than using the Project window.

Additionally to the scenes of the game, most of the downloadable Unity Assets come with example scenes to test them.

3.5.2 Effects

The graphical effects are the core of the game. Unity allows the possibility to add these effects by attaching components that extend the *PostEffectsBase* class. These components read the texture generated by the screen, that is, an array of color pixels, and apply a transformation. Notice that some effects like blur and FOV were easy to implement, but others required additional investigation and self-implementation.

The classes that manage the effects are *CameraEffect* and its 8 subclasses, *EffectData* and *EffectsController*. Figure 3.33 shows the class diagram for effects implementation.

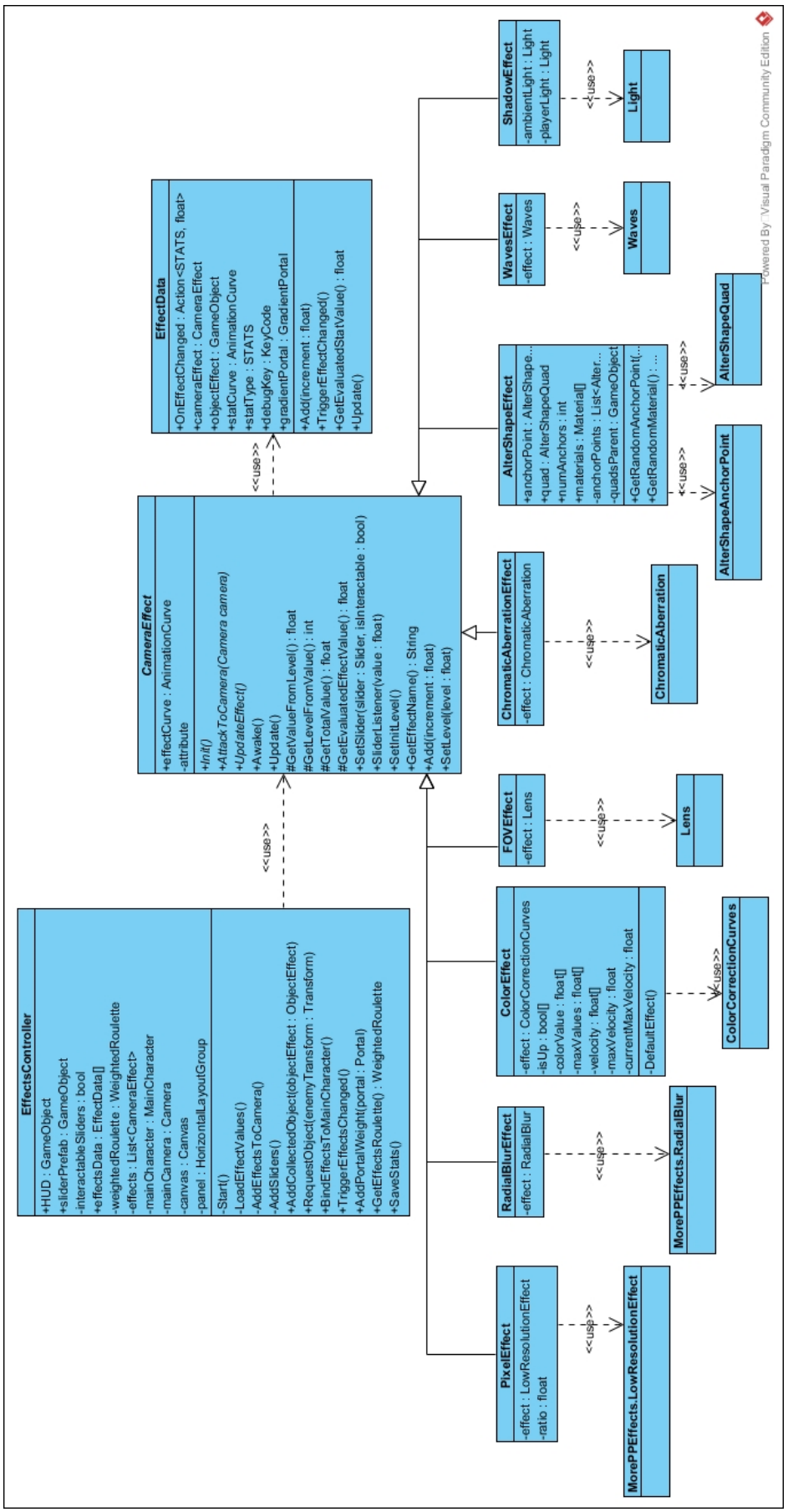


Figure 3.33: Effects class diagram

CameraEffect

The *CameraEffect* is the component that manages the state of an effect applied to the player. It contains an *AnimationCurve* to know how it changes, some adaptation variables (values and levels), the name and color of the effect and the slider used in the GUI. All the effects derive from this abstract class, so every subclass must implement the *Init*, *AttachToCamera* and *UpdateEffect* functions because each one has a different behavior and additional variables. The *value* variable is in the [0,1] range and it is used to evaluate the *StatCurve* to know the stat for the main character. *minValue* and *maxValue* are the bounds for the implemented effect. For example, *ChromaticAberration* goes from 0 to 20, being 0 no effect, and 20, the maximum color vibration decided. *minLevel* and *maxLevel* are 0 and 10. They are used to obtain a better perception in the game than the [0,1] range from *value*.

On the *Init* function every effect implements the additional logic (if it exists) to make the effect work. It acts as a hook. *AttachToCamera* requires the main camera to add the effect component to it. Most of the effects only do the attach, but others requires special initializations after the effect is applied. *UpdateEffect* does the calculations to adapt the value variable in the range of [minValue, maxValue] to the behavior of the effect at that point.

PixelEffect

The *LowResolutionEffect* shader from *More Post-Processing Effects* asset is used. It only requires the screen ratio to maintain a correct aspect ratio:

```
1 override
2 protected void Init ()
3 {
4   ratio = ((float)Screen.width) / Screen.height;
5 }
```

The resolution of the effect (and hence, the screen) is set:

```
override
protected void UpdateEffect () {
        effect.resolutionX = (int)GetEvaluatedEffectValue();
        effect.resolutionY = (int)(effect.resolutionX / ratio);
}
```

RadialBlurEffect

The *RadialBlur* shader from *More Post-Processing Effects* asset is used. It doesn't require additional data. *UpdateEffect* has all the necessary. When the game is paused, the effect acts, so it is deactivated in the pause menu. Additionally to the *blurStrength*, to obtain the desired result, the starting point of the effect is where the mouse is located:

```
override
protected void UpdateEffect () {
  if(PauseLogic.isGamePaused())
  {
    return;
  }
  effect.blurStrength = GetEvaluatedEffectValue();
  effect.centerX = Input.mousePosition.x / Screen.width;
  effect.centerY = Input.mousePosition.y / Screen.height;
    }
```

ColorEffect

Color effect uses the *ColorCorrectionCurves* component to alter the RGB and saturation of the camera. It has an adjusted configuration to move the curves with more sense:

```
// Saturation, red, green, blue
float[] colorValue = { 1f, 0f, 0f, 0f };
    float[] maxValues = { 5f, 1f, 1f, 1f };
    float[] velocity = { 0.32f, 0.59f, 0.37f, 0.17f };
    float maxVelocity = 4f;
```

At each frame the endpoints of the curves are moved up or down, depending on the *isUp* array, with a random velocity. Once one curve reach the bottom or the top, a new velocity is assigned and the direction is reversed.

```
override
protected void UpdateEffect()
{
  float currentMaxVelocity = GetEvaluatedEffectValue();
  float currentLevelValue = currentMaxVelocity/2;

  if(value < 0.3f)
  {
    DefaultEffect();
    return;
```

```
  }

  for (int i = 0; i < 4; ++i)
  {
    if (isUp[i])
    {
      colorValue[i] += Time.deltaTime * velocity[i];
      if (colorValue[i] >= maxValues[i] * currentLevelValue)
      {
        isUp[i] = false;
        velocity[i] = Random.Range(currentLevelValue, currentMaxVelocity);
      }
    }
    else {
      colorValue[i] -= Time.deltaTime * velocity[i];
      if (colorValue[i] <= 0)
      {
        isUp[i] = true;
      }
    }

    switch (i)
    {
      case 0:
        effect.saturation = colorValue[i];
        break;
      case 1:
        effect.redChannel.MoveKey(0, new Keyframe(0, colorValue[i]));
        effect.redChannel.MoveKey(1, new Keyframe(1, 1f - colorValue[i]));
        break;
      /*Omitted other of colors*/
    }
  }
  effect.UpdateParameters();
}
```

FOVEffect

The *Lens* shader from *More Post-Processing Effects* asset is used. The effect is simply updated using the corresponding value:

```
override
protected void UpdateEffect()
{
  effect.lensDistortion = GetEvaluatedEffectValue();
}
```

ChromaticAberrationEffect

Unlike the rest of the effects that use shaders, for this effect no one of the available shaders on Internet convinced me. So I implemented my own shader. The shader receives 6 floats, corresponding to the movements of the 3 RGB colors on each side (X and Y) and applies movements over each color layer. Although the code is basic, since the shader programming language was unknown for me, I had to learn the syntax and available methods:

```
1  Shader "Custom/ChromaticAberration" {
2  Properties{
3    _MainTex("Base (RGB)", 2D) = "white" {}
4  }
5    SubShader
6  {
7    Pass
8  {
9
10   CGPROGRAM
11   #pragma vertex vert_img
12   #pragma fragment frag
13   #pragma fragmentoption ARB_precision_hint_fastest
14   #include "UnityCG.cginc"
15
16   uniform sampler2D _MainTex;
17   uniform float _AberrationOffsetRedX;
18   uniform float _AberrationOffsetRedY;
19   /* Ommited other colors */
20
21   float4 frag(v2f_img i) : COLOR
22   {
23
24     float2 coords = i.uv.xy;
25
26     _AberrationOffsetRedX /= 360.0f;
27     _AberrationOffsetRedY /= 360.0f;
28     /* Ommited other colors */
29
30     //Red Channel
31     float2 redCoords = coords.xy;
32     redCoords.x = redCoords.x - _AberrationOffsetRedX;
33     redCoords.y = redCoords.y + _AberrationOffsetRedY;
34     float4 red = tex2D(_MainTex , redCoords);
35     /* Ommited other colors */
36
37     float4 finalColor = float4(red.r, green.g, blue.b, 1.0f);
38     return finalColor;
39   }
40
41   ENDCG
```

```
42
43      }
44    }
45  }
```

The code to manage the rotation of each color layer was adapted from the project *Fruit Slicing Game* [13]. The *UpdateEffect* function sends the data to this script:

```
1   override
2     protected void UpdateEffect()
3     {
4         float currentValue = GetEvaluatedEffectValue();
5         effect.ChromaticAberrationRedX = Random.Range(0f, currentValue);
6         effect.ChromaticAberrationRedY = Random.Range(0f, currentValue);
7         /* Ommited other colors */
8     }
```

AlterShapeEffect

Alter shape requires additional components. First, for the initialization, an ellipse of anchor points is created around the camera. The *AnchorShapeAnchorPoint* class only contains the number position. When the effect requires a quad, a *AlterShapeQuad* is created. This class manages the rotation, direction and speed of the quads displayed on the screen. Since the anchor points are attached to a camera, and the game contains two visualization modes, every camera requires its own ellipse and quad sizes.

The *UpdateEffect* function creates or deletes the quads when necessary:

```
1   override
2     protected void UpdateEffect()
3     {
4         float currentValue = GetEvaluatedEffectValue();
5         int expectedNumQuads = (int)currentValue;
6         int numQuads = quadsParent.transform.childCount;
7
8         if (expectedNumQuads < numQuads)
9         {
10            Destroy(quadsParent.transform.GetChild(numQuads-1).gameObject);
11        }
12
13        if (expectedNumQuads > numQuads)
14        {
15            AlterShapeQuad currentQuad = Instantiate(quad);
16            currentQuad.transform.SetParent(quadsParent.transform, false);
17            currentQuad.alterShapeManager = this;
```

```
18              currentQuad.Init(anchorPoints[Random.Range(0, anchorPoints.Count)], 3f,
                ↪  100f, SaveData.GetInstance().userOptions.cameraType ==
                ↪  CameraType.ORTOGRAPHIC ? 3f : 1f);
19          }
20      }
```

When an *AlterShapeQuad* is created, it requests a material to the *AltherShapeEffect* (there are 10 materials, each one is a basic color) and a new anchor point to go. The new anchor point is decided according to the next code. Every quad must know where to go due to when it reaches an anchor, the next movement is selected depending on this:

```
1  int maxMovements = 2;
2  int newPosition = (lastPosition + anchorPoints.Count / 2) +
   ↪  Random.Range(-maxMovements, maxMovements);
```

WavesEffect

The *Waves* shader from *More Post-Processing Effects* asset is used. As other effects, the evaluated value is sent to the effect:

```
1    override
2      protected void UpdateEffect()
3      {
4          float value = GetEvaluatedEffectValue();
5          effect.strengthX = value;
6          effect.strengthY = value;
7      }
```

ShadowEffect

Shadow effect uses two lights, the scene ambient light and the light attached to the main character. There is no attachment to the camera because it does not affect that affects the screen. The higher the value is, the more light is generated by the player and the less ambient light is there:

```
1    override
2      protected void UpdateEffect()
3      {
4          float currentValue = GetEvaluatedEffectValue();
5          playerLight.intensity = currentValue;
6          ambientLight.intensity = StaticTools.Map(maxValue -
           ↪  GetEvaluatedEffectValue(), minValue, maxValue, 0f, 1f);
7      }
```

EffectData

EffectData works as a layer between the main character and his effect stats, that use a *CameraEffect*. It contains an *AnimationCurve* that evaluates the attached stat value, the corresponding object type dropped by an enemy when it is requested, and the spawned portal type. Every *EffectData* is attached to the *EffectsController* so that when an effect must change, the *EffectsController* notifies *EffectData*, and *EffectData* updates the *CameraEffect* value and notifies the main character. Finally the main character requests which the new stat effect value is.

EffectsController

This Singleton class coordinates the initialization of effects, their sliders, the triggers for the *MainCharacter* component and the values of the *EffectData* components. Finally, it contains the logic of a weighted roulette to spawn effects objects when an enemy dies.

3.5.3 Combat system

Introduction

The game combat system requires a perfect coordination among a lot of components. This system must be isolated, working be itself and allowing reusable classes to implement all the interactions among the characters. Figure 3.34 shows the class diagram of the combat system.

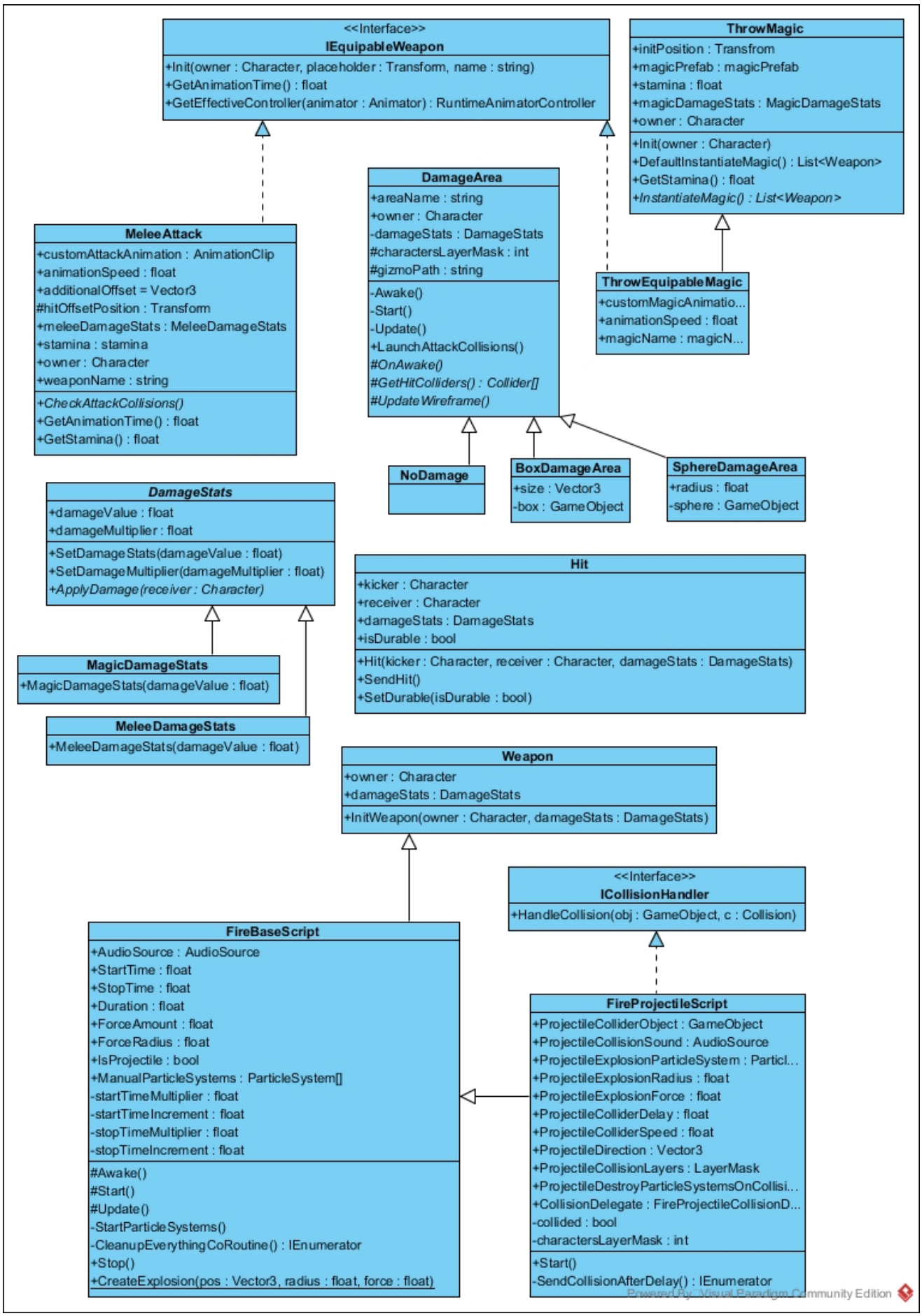


Figure 3.34: Combat system class diagram

Hit

The most basic class is Hit. It contains the logic of a collision between two characters. Currently there is only one type of hit. But extending it, we can obtain other effects, such as, a combination of melee and magic hits, durable hits directly attached to the receiver with self-management (i.e. applying a decreasing damage instead of constant), environmental damage, modification of the main player stats (i.e. overload the stamina impeding he can attack) or even graphical effects. The possibilities are wide.

DamageStats

This class stores the damage of an attack. The *MainCharacter* has both subclasses to store the melee weapon and the magic damages. For the enemies, every attack has its own *DamageStats* related to its *DamageArea* or *ThrowMagic*. When the damage is to be applied, the receiver gets it from the kicker, but the receiver also applies its own modifiers altering the final received damage. This is the implementation for magic:

```
1 public override void ApplyDamage(Character receiver)
2 {
3   receiver.ReceiveMeleeDamage(damageValue*damageMultiplier);
4 }
```

IEquipableWeapon

This interface defines the common functions of *MeleeAttack* and *ThrowEquipableMagic*. The classes allow the player attack with different kind of weapons and magic spells. Both classes share some variables, but *ThrowEquipableMagic* extends from *ThrowMagic*.

The common functionalities are the usage of a custom animation, the speed of the animation, the use of stamina, an initialization of prefabs in the corresponding hand of the the main character and finally the override of the custom animation when the weapon changes.

ThrowMagic

ThrowMagic is the class used by the components of the characters that cast magic spells. It contains the Transform defining where the magic appears, the magic spell prefab, its magic damage statistics

and the creator of the spell. There is also the stamina, used principally by the main character, but also can be used for custom cooldown implementations. Every *ThrowMagic* component must be initialized by the owner and the *DamageStats* component must be present in the same object. As an example and being the default behavior of the component, it is showed how a magic spell is instantiated. All the prefabs must contain a Weapon component or the process will fail. A list of Weapons is used since an spell can spawn more than one instance:

```
1   public List<Weapon> DefaultInstantiateMagic ()
2     {
3         GameObject magicGameObject = Instantiate(magicPrefab, initPosition.position,
          ↪  initPosition.rotation) as GameObject;
4         Weapon magic = magicGameObject.GetComponent<Weapon>();
5         magic.InitWeapon(owner, magicDamageStats);
6         List<Weapon> magicWeapons = new List<Weapon>();
7         magicWeapons.Add(magic);
8         return magicWeapons;
9     }
```

Every subclass of *ThrowMagic* must implement the abstract function *InstantiateMagic*, that returns the list of spawned spells. By default no more logic is required, but if the spells require further initialization, the subclass must do it.

ThrowEquipableMagic

The *ThrowEquipableMagic* component manages the logic between the *MainCharacter* component and the capacity to cast magic spells provided by *ThrowMagic*. In the initialization the owner is set, the component is moved to the placeholder *Transform* (being the left hand), the starting point of the spells is stored and finally the previous animation and velocity for magic is overridden with the animation carried by the spell. Every *ThrowEquipableMagic* component has a particle system representing the current spell (normally a flame in the left hand).

MeleeAttack

MeleeAttack works exactly as *ThrowEquipableMagic* excepting that instead of use *InstantiateMagic*, it uses *CheckAttackCollisions* as attack function. It also manages the trail of the blade (managed by *MeleeWeaponTrail* component created by Anomalous Underdog[67]). In this case the particle system is directly replaced by the weapon, being a mesh. Most of the attacks are implemented as follows.

When the animation event is fired, the sound of the weapon is played, then it is checked if inside the area of attack there are objects with the character mask. It that case, it is checked to avoid self-damage. Finally, a hit is sent to every affected Character component (all objects with Character layer must present a *Character* implementation) with the corresponding damage stats:

```
1   public override void CheckAttackCollisions()
2     {
3         GetComponent<AudioSource>().Play();
4         Collider[] hitColliders = Physics.OverlapSphere(hitOffsetPosition.position,
          ↪  radius, charactersLayerMask);
5
6         foreach (Collider currentCollider in hitColliders)
7         {
8             if (gameObject.layer != currentCollider.gameObject.layer)
9             {
10                Hit hit = new Hit(this.owner,
                  ↪  currentCollider.gameObject.GetComponent<Character>(),
                  ↪  meleeDamageStats);
11                hit.SendHit();
12            }
13        }
14    }
```

DamageArea

As a simplification of the physics of the combat system and to avoid problems, the hit boxes for the enemy attacks are implemented in a similar way as it is done done for the main character melee weapons.

A *DamageArea* stores a region where the enemies can perform attacks using Unity Physics. It can also be used for environmental damages as fire, poison or other kinds of danger.

Every implementation of *DamageArea* must extend this abstract class. There are implemented box areas, sphere areas, and also no damage areas as a *Nullable* objects.

The *LaunchAttackCollisions* works in the same way as the *MeleeAttack*. With the particularity now *GetHitColliders* must be implemented, for example, the box implementation is:

```
1   protected override Collider[] GetHitColliders()
2   {
3     return Physics.OverlapBox(this.transform.position, size/2,
      ↪  this.transform.localRotation, charactersLayerMask);
4   }
```

For debug purposes, a *DamageArea* can also be displayed as a red figure to check everything is working properly.

3.5.4 Character hierarchy

Figure 3.35 shows the class diagram for the characters implementation.

The abstract class Character groups the common behaviors and variables that every character in the game has. A Character has the maximum health, the current health, a score to know its importance, the percentage of damage absorption for melee and magic attacks, a boolean to know if it is alive or not, and its name. There are also two Action callbacks to know when a character is hit or damaged.

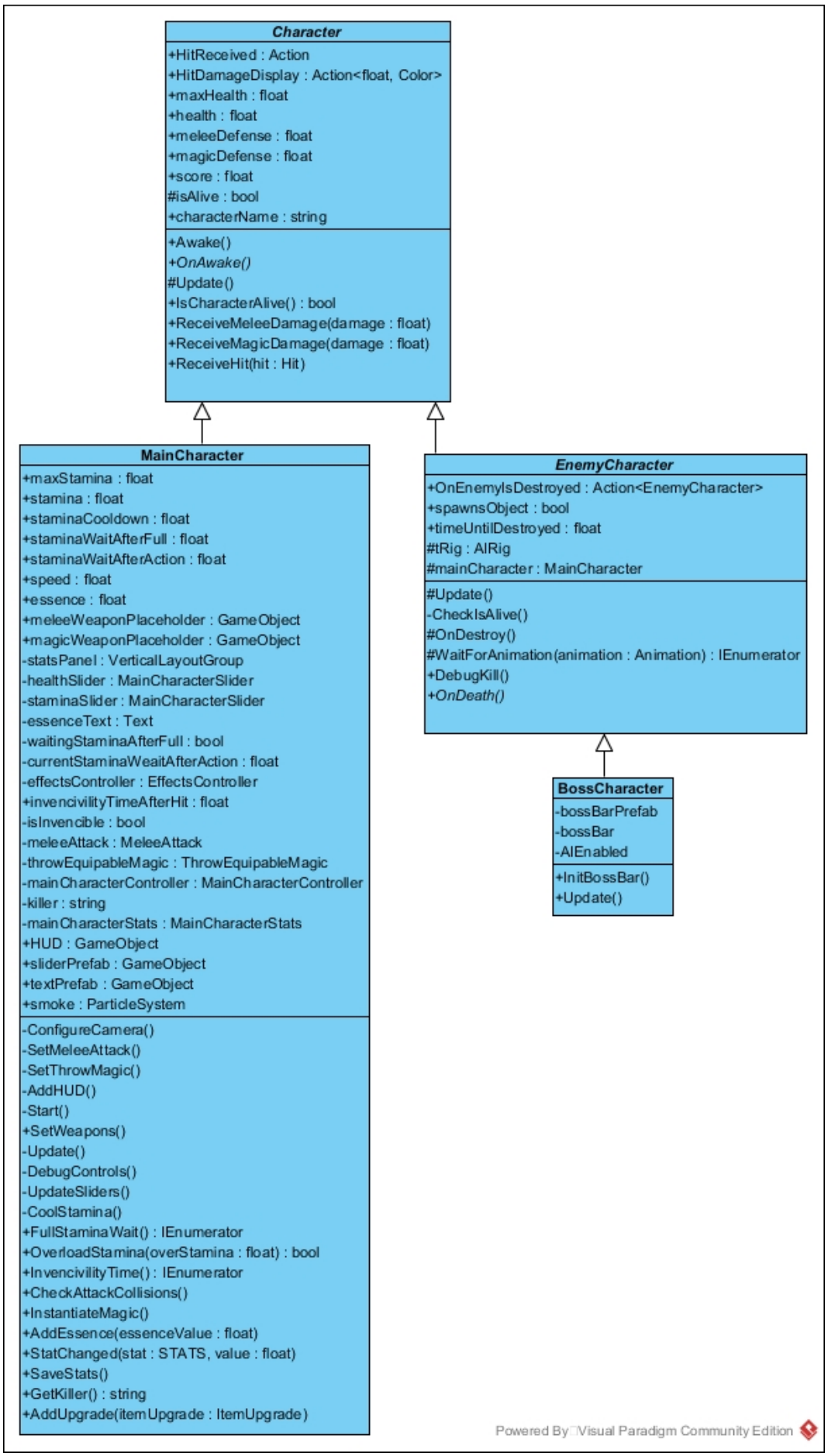


Figure 3.35: Character hierarchy class diagram

When a character is spawned, its health is set to the maximum and marked as alive. There is also an additional OnAwake hook function for its subclasses. At every frame the enemy is checked to be is alive. When a Character sends a Hit to another Character, the receiver executes the following code:

```
1   public virtual void ReceiveHit(Hit hit)
2     {
3         hit.damageStats.ApplyDamage(this);
4         if(HitReceived != null)
5         {
6             HitReceived();
7         }
8     }
```

At the end of the events chain to apply the damage, ReceiveMagicDamage or ReceiveMeleeDamage are called depending on the damage type. As an example we present the implementation for magic. The absorption of damage of the character is calculated, the health is updated and the HitDamageDisplay action for the DamageDisplay component is triggered, sending the quantity of damage and a color, blue due to it being magic (red for melee):

```
1   public void ReceiveMagicDamage(float damage)
2     {
3         float realDamage = damage * (1 - magicDefense);
4         health = Mathf.Max(0, health - realDamage);
5         if (HitDamageDisplay != null)
6         {
7             HitDamageDisplay(realDamage, Color.blue);
8         }
9     }
```

The rest of functionalities must be implemented in the subclasses.

Main character

The MainCharacter component manages the stats and player HUD. It also coordinates the rest of related components like weapons usage and controls.

When a level starts (in the Awake phase), this component attaches the health and stamina bars to StatsPanel, essence text to the EssencePanel and every effect bar to BarPanel. All the panels are Layout groups. There are some graphical issues to obtain the same graphics regardless of the screen

resolution. The other step performed is the configuration of the camera, being it orthographic or behind the character.

At the Start phase, the EffectsController sets the weapons to the main character. This is done because the bind of effects must be done before. Otherwise the damage stats will fail when they found there is no data to evaluate the damage. In the same phase, the health and essence are set from StaticData.mainCharacterInfo. By default stamina starts at 0.

The Update function calls base. Update to maintain the inherited functionalities. It checks cheats (max health and 0 stamina ever) and debug functions. It also updates the values of the health and stamina sliders and cools the stamina. The functions CoolStamina, FullStaminaWait and OverloadStamina (adds stamina) manage the value of the stamina, including the smoke at the left shoulder of the main character.

In this case the ReceiveHit function is overridden since main character has invincibility time after a hit, but durable hits are not affected. To know who the killer is, at every Hit received, the owner of the attack is stored. So finally when player dies the last enemy that attack the player becomes the real killer:

```
1  public override void ReceiveHit(Hit hit)
2      {
3          if(!isInvencible || hit.isDurable)
4          {
5              hit.damageStats.ApplyDamage(this);
6              killer = hit.kicker.characterName;
7              StartCoroutine(InvencibilityTime());
8              if (HitReceived != null)
9              {
10                 HitReceived();
11             }
12         }
13     }
```

Finally, every time an effect stat is updated, StatChanged receives the event. The change to the mainCharacterStats instance is sent and the new value is reevaluated (a stat value is composed by an effect stat and an upgrade stat), and the necessary changes are performed:

```
1  public void StatChanged(STATS stat, float value)
2      {
3          mainCharacterStats.SetEffectStat(stat, value);
4
```

```
5           switch(stat)
6           {
7               case STATS.HEALTH:
8                   health = AdaptCurrentValueWhenIncreasesMaxStat(health, maxHealth,
                    ↪  mainCharacterStats.Get(stat));
9                   maxHealth = mainCharacterStats.Get(stat);
10                  healthSlider.slider.maxValue = maxHealth;
11                  healthSlider.UpdateText();
12                  break;
13              case STATS.SPEED:
14                  speed = mainCharacterStats.Get(stat);
15                  mainCharacterController.SetSpeed(speed);
16                  break;
17              case STATS.MELEE_ATTACK:
18                  meleeAttack.meleeDamageStats.
19                  SetDamageMultiplier(mainCharacterStats.Get(stat));
20                  break;
21              case STATS.MELEE_DEFENSE:
22                  meleeDefense = Mathf.Min(0.9f, mainCharacterStats.Get(stat));
23                  break;
24              case STATS.MAGIC_ATTACK:
25                  throwEquipableMagic.magicDamageStats.
26                  SetDamageMultiplier(mainCharacterStats.Get(stat));
27                  break;
28              case STATS.MAGIC_DEFENSE:
29                  magicDefense = Mathf.Min(0.9f, mainCharacterStats.Get(stat));
30                  break;
31              case STATS.STAMINA:
32                  stamina = AdaptCurrentValueWhenIncreasesMaxStat(
33                  stamina, maxStamina, mainCharacterStats.Get(stat));
34                  maxStamina = mainCharacterStats.Get(stat);
35                  staminaSlider.slider.maxValue = maxStamina;
36                  staminaSlider.UpdateText();
37                  break;
38              case STATS.STAMINA_INCREASE:
39                  staminaCooldown = mainCharacterStats.Get(stat);
40                  break;
41          }
42      }
```

MainCharacterStats

This class supports the MainCharacter component by encapsulating the logic about stats. It contains an array for the effect stats, another for the upgrade stats and a list to store all the upgrades the player obtained. Every change over the Main Character stats must be sent to this class to obtain the real stat values. Since it is Serializable, the save system also uses it to store and load the stats of the player.

MainCharacterController

This controller is a modified version of the original RPGCharacterControllerFREE component from Warrior Pack Bundle 2 FREE by Explosive[21]. The original code has the necessary behaviors to move the character and perform the melee attack. The following code shows a little part of the logic. The two input directions are read, modeled as a Vector3, and depending on the current state, some variables of the animation state machine are activated. For attacks and dash, if the state machine is ready to perform it and there is enough stamina, the action is executed by blocking the capacity to perform more until it is finished:

```
1  //Get input from controls
2  float z = Input.GetAxisRaw("Horizontal");
3  float x = -(Input.GetAxisRaw("Vertical"));
4  inputVec = new Vector3(x, 0, z);
5
6  if (x != 0 || z != 0)  //if there is some input
7  {
8    //set that character is moving
9    animator.SetBool("Moving", true);
10   animator.SetBool("Running", true);
11   animator.SetFloat("SpeedFactor", 1f);
12   }
13   else
14   {
15     //character is not moving
16     animator.SetBool("Moving", false);
17     animator.SetBool("Running", false);
18   }
19
20   if (Input.GetButtonDown(Controls.MAGIC_BUTTON) && usesStamina &&
     ↪  mainCharacter.OverloadStamina(throwMagic.GetStamina()))
21   {
22     animator.SetTrigger("Attack1Trigger");
23     StartCoroutine(COStunPause(throwMagic.GetAnimationTime()));
24   }
```

The animation state machine is presented in Figure 3.36. Figure 3.37 shows an example of transition and Figure 3.38 a 1-dimensional Blend Tree to set the direction of the run animation.

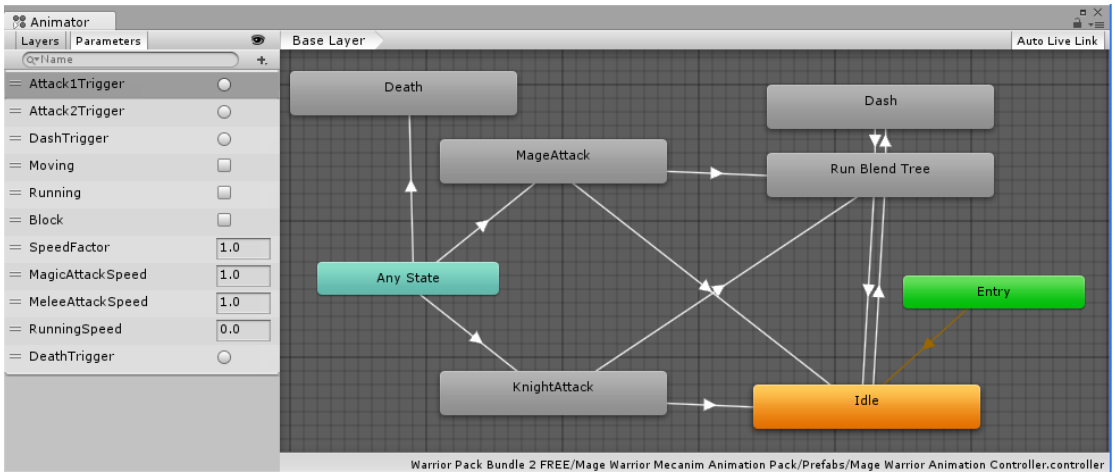


Figure 3.36: Main character animation state machine

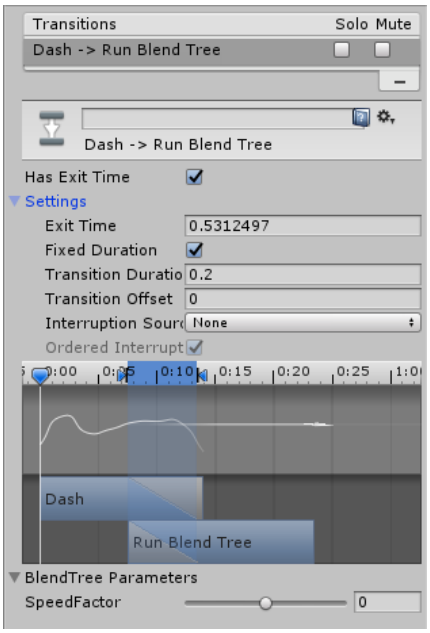


Figure 3.37: Dash to run transition

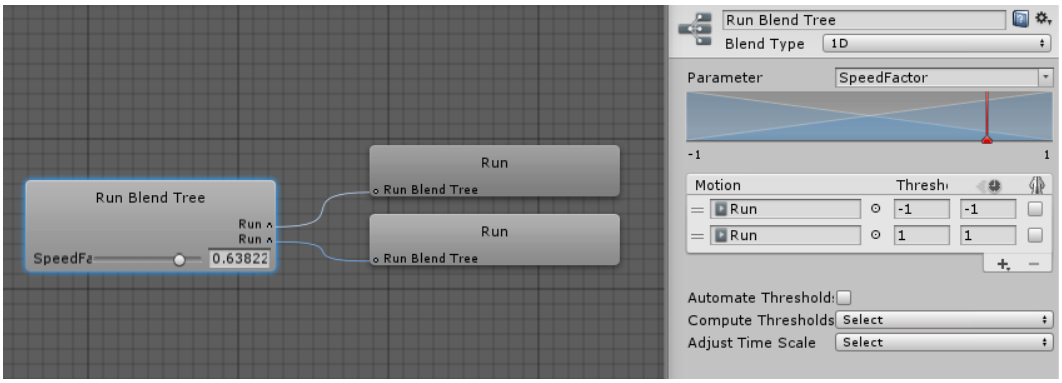


Figure 3.38: Main character run blend tree

Initially the dash feature was very inaccurate. It just changed the character speed. This provoked lot of collision problems, including walls and enemies crossing. This was solved by means of a raycast implementation, using Math.Lerp to move the main character to the final position (the grey box) in a given time. In the case the ray collides with another object, the main character will never cross it. The Figures 3.39 and 3.40 show how the ray limits the dash range.

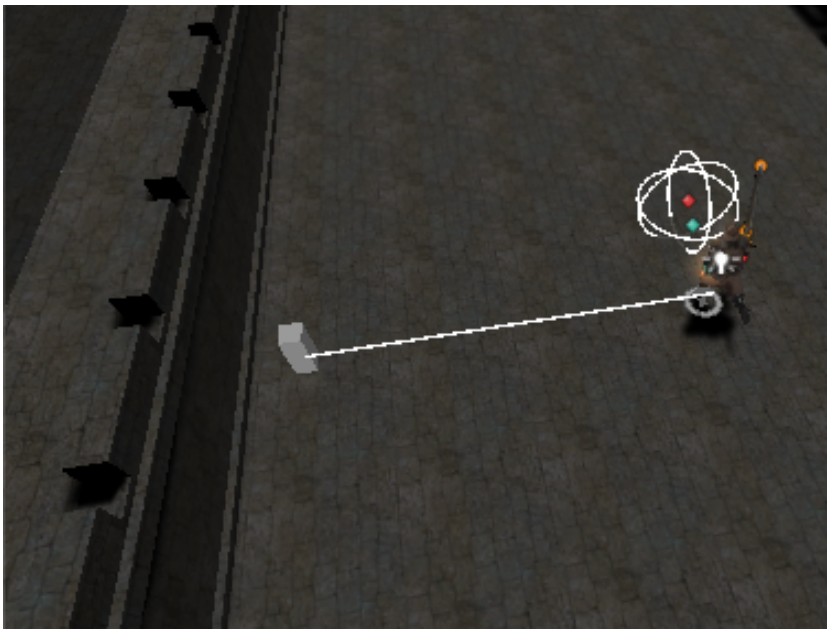


Figure 3.39: Complete dash

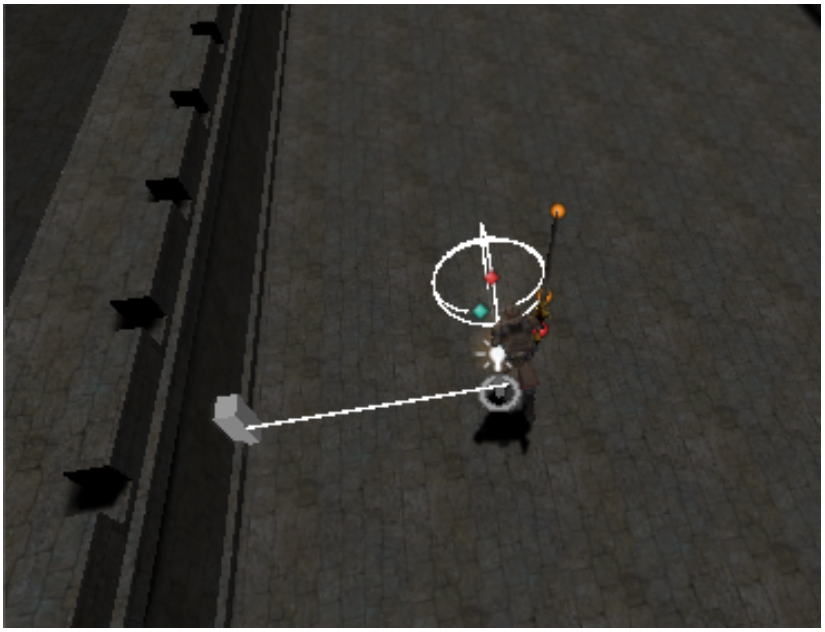


Figure 3.40: Dash stopped by wall

The rotation of the main character, ever facing the mouse, was implemented by adapting the logic

from Survival Shooter tutorial by Unity Technologies[63]. In this project there is a Mesh Collider at the height of character's gun. At each frame, a RaycastHit is launched from the camera to the adapted world position of the mouse. Once the collision point is obtained,it applies the corresponding rotation to the Rigidbody of the character.

Finally, the controls of the main character can be blocked or unblocked when required, for example, when the player reaches a portal.

3.5.5 Game loop

To get a better view of the components used in key moments of game operation, the following sequence diagrams are presented.

At the *Awake* function (Figure (3.41) the initialization and binding of components are performed.

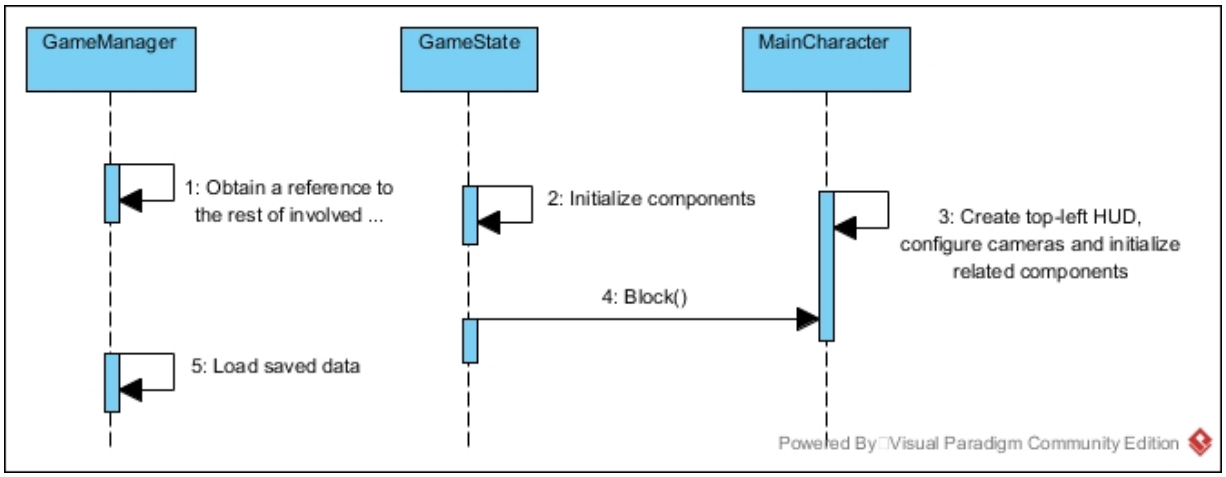


Figure 3.41: Awake sequence diagram

Start function (Figure 3.42) contains the rest of the initializations that require the previous data created in *Awake* functions. The most important step is the asynchronous generation of the map.

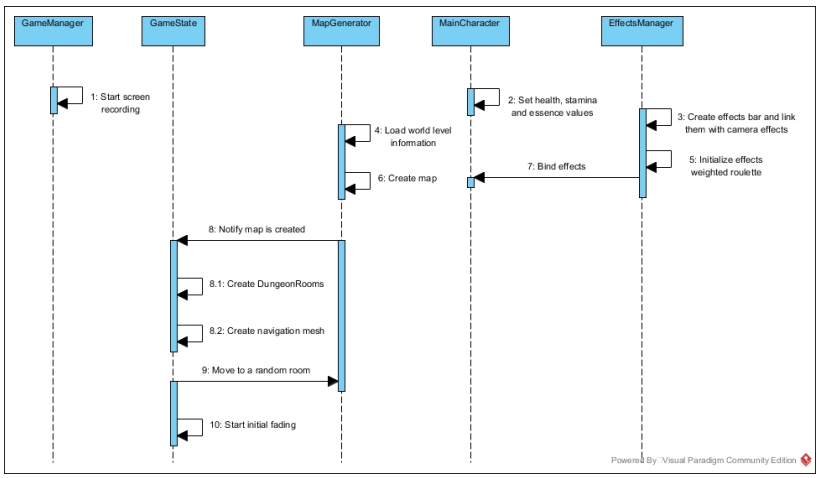


Figure 3.42: Start sequence diagram

In general, the components of the game (characters, rooms, items) manage its own life cycle. Most of the changes are notified using the subscription system provided by C#. That is *GameState* is the only component that constantly checks if the game must continue using the *Update* function (Figure 3.43).

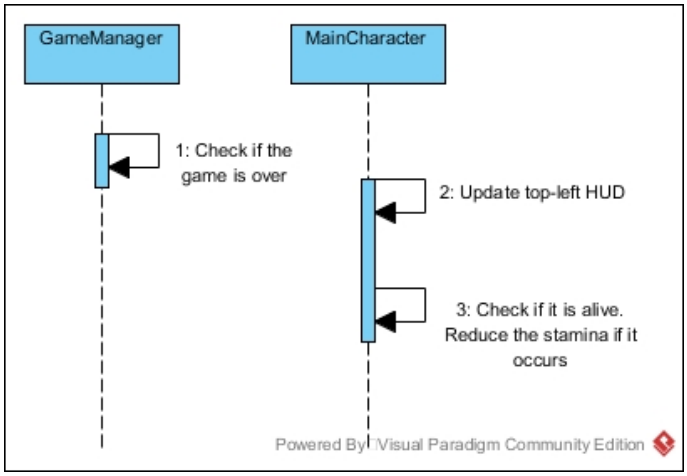


Figure 3.43: Update sequence diagram

When player enters on a room, the collision that envelops it notifies *GameState* to fill it with enemies.

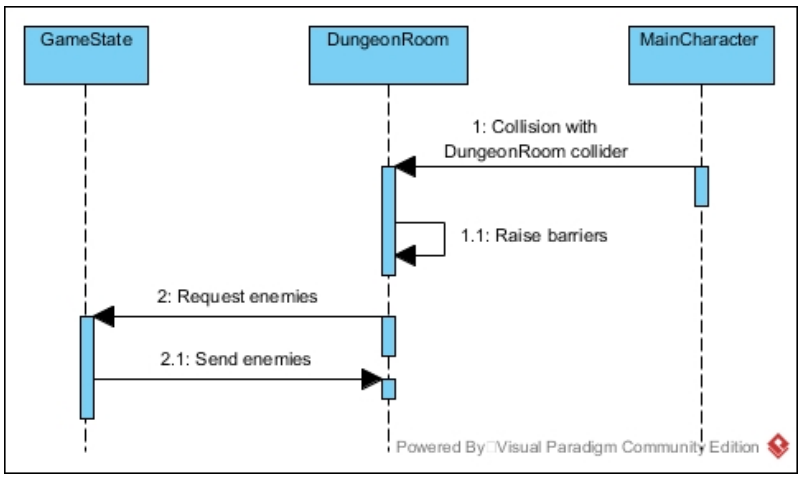


Figure 3.44: The player enters a room sequence diagram

DungeonRoom receives the death event of each enemy in the room. When all the enemies are defeated, *DungeonRoom* lowers the barriers and asks to *GameState* if a portal must be spawned. If it occurs, *GameState* obtains the portal type from *PortalManager* and spawns it in the *DungeonRoom* that requested the portal.

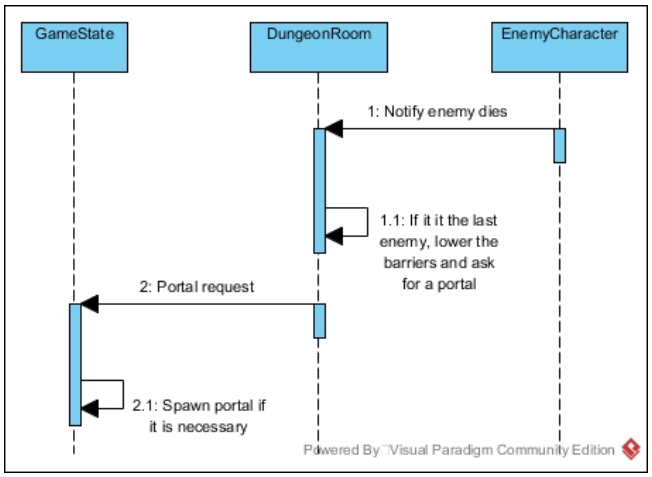


Figure 3.45: The player finishes a room sequence diagram

Finally, when player enters in a portal, the level is marked as finished and the ending fade appears. The player controls are blocked, the portal type is added to the effects weighted roulette and the information of the run (defeated enemies and time) is added to the run data. Finally, the data is saved to maintain the statistics for the following scenes.

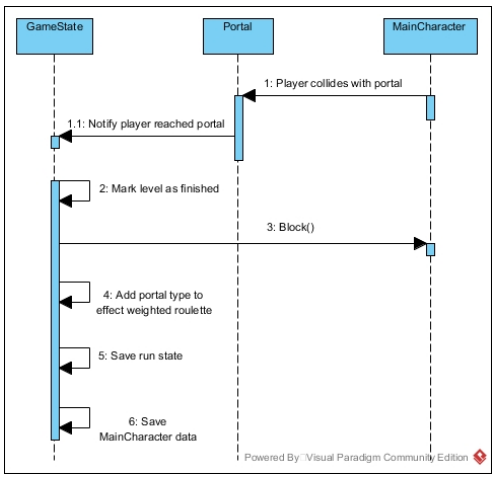


Figure 3.46: The player enters a portal sequence diagram

3.5.6 Enemies

EnemyManager

When a level is loaded, EnemyManager obtains the information about the world by loading the right EnemyWorldContainer, which contains the maximum score and a list with the enemies, each with assigned score. A WeightedRoulette is filled with this information. When the enemies are required, the sum of their scores can't overcome the maximum score.

When MainCharacter enters an unvisited room, GameState provides the enemies for this. First, the available floor quads are obtained: those are empty floors and floors further than 10 distance units from the player to avoid unfair situations. The obtained list is shuffled. Then, the next formula defines the number of enemies:

$$\#enemies\ in\ room = \min(\#\ enemies\ in\ world, \max(3, valid\ floors\ \cdot\ 0.025))$$

Finally, a floor is assigned to each enemy. The resulting list is returned to GameManager.

RAIN AI

RAIN AI was used to allow the enemies to detect and pursue the player.Figure 3.47 shows the script used by EnemySpring and Figure 3.48 shows its configuration. The behavior is split in two actions:

- parallel: All the actions are executed at the same time. *Detect* uses the specified radius in the sight sensor to know the position of MainCharacter. If it is found, the enemy uses the navigation map to reach MainCharacter.
- attack: The AI uses another sight sensor to detect when it can attack. It stores the result in the *canAttack* variable.

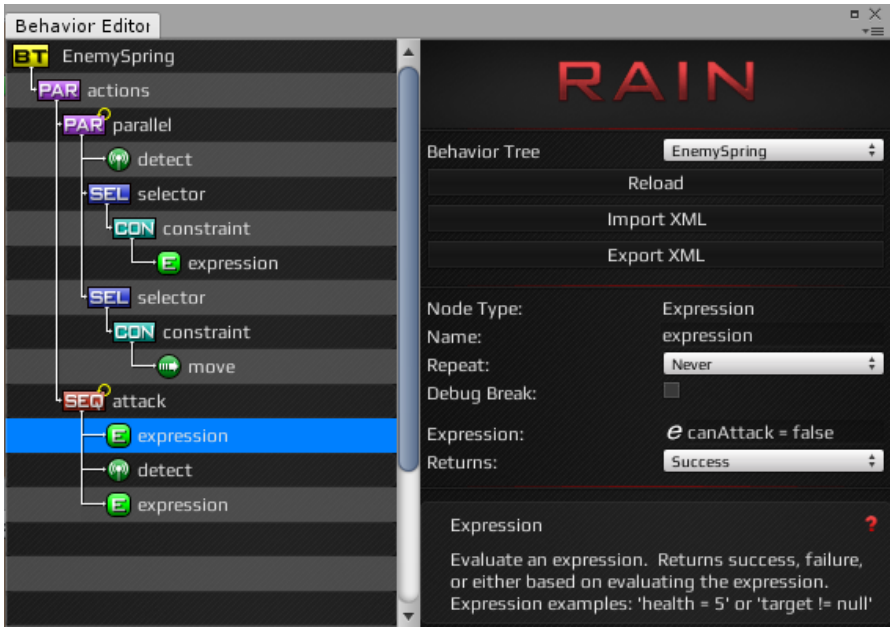


Figure 3.47: Rain AI script for Enemy Spring

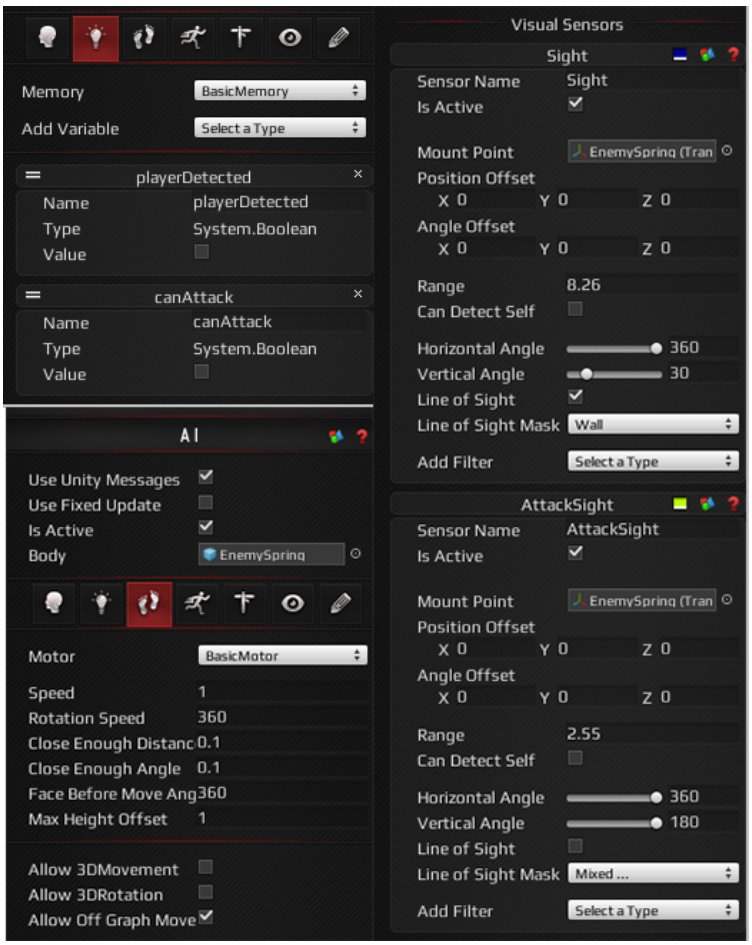


Figure 3.48: AI Rig configuration for Enemy Spring

The flow diagram depicked in Figure 3.49 shows the actions performed by the script. Notice that if an action has no end, like *move* (in green), the AI will repeat this action indefinitely. The other actions are available due to the parallel environment. Finally, the variables are accessible in Unity after the completion of a sequence of actions. *canAttack* is accessible in Unity before it changes again to false.

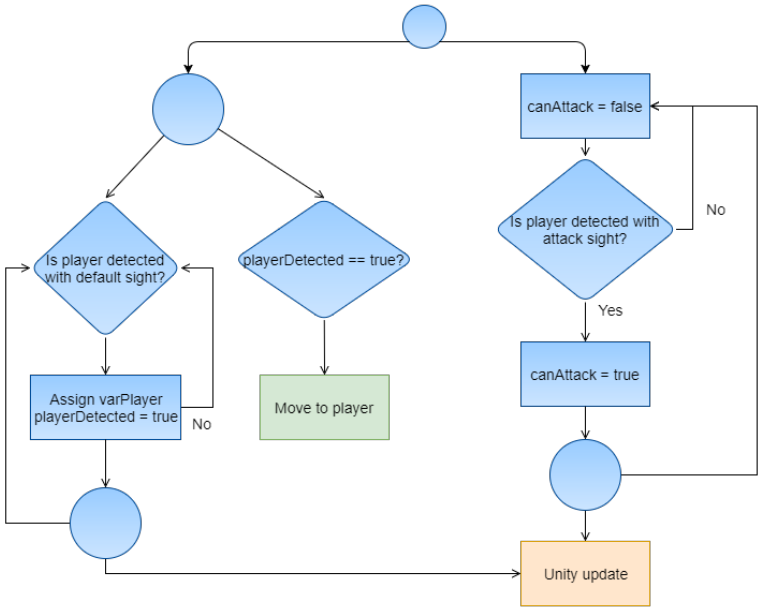


Figure 3.49: Enemy Spring Rain AI flow diagram

After learning how RAIN AI works and implementing some enemies, its use was finally discarded in favor of a more comfortable solution. The reasons to decide the creation of a new AI system are the next enumerated:

1. Difficulties in the communication between RAIN AI and Unity scripts
2. Low efficiency due to constant checks to obtain updated variables
3. Difficulties to debug
4. Chained actions are treated as independent states, making the code/action tree difficult to trace
5. Poor documentation
6. Broken functionalities like the line of sight

Custom AI

The new AI system was designed as an asynchronous generic pattern design compatible with all the enemies and bosses of the game.

The system works in the following way:

- All the actions must be contained in an *enum*, for example:

```
protected enum EnemyLightbombState { IDLE, WAIT, PATROL, PURSUE, EXPLODE,
↪ HEAD_ATTACK, SHOT, DEATH }
```

- The entity that uses the AI system must implement the following functions: SetAIState and AIAction. The first method assigns the a state. The second one launches the selected action:

```
protected virtual IEnumerator AIAction()
    {
        idle = false;
        yield return StartCoroutine(AsyncDecideAction());
        idle = true;
    }
```

- Since the system is asynchronous, all the functions must return an IEnumerator and they must be called using StartCoroutine:

```
protected IEnumerator AsyncDecideAction()
    {
        switch (state)
        {
            case EnemyLightbombState.WAIT:
                yield return StartCoroutine(AsyncWait());
                break;
            case EnemyLightbombState.PATROL:
                yield return StartCoroutine(AsyncPatrol());
                break;
            // Omitted other actions
            case EnemyLightbombState.SHOT:
                yield return StartCoroutine(AsyncShot());
                break;
            case EnemyLightbombState.DEATH:
                yield return StartCoroutine(AsyncDeath());
                break;
        }
    }
```

- Every action implements a behavior that can be composed of several subactions. Thanks to the coroutines, the code is clear and doesn't require to jump to several independent actions as it was the case with RAIN AI. For example:

```
IEnumerator AsyncShot()
    {
        isDoingAction = true;
        yield return new WaitForSeconds(0.5f); // Wait some time to start the
        ↪ animation
        ClearAnimationStates();
        animator.SetTrigger("shot"); // Animation started
        yield return new WaitForSeconds(0.4f); // Wait until the enemy is in
        ↪ the correct animation frame
        gun.InstantiateMagic(); // Shots
        yield return new WaitForSeconds(0.2f); // Wait until animation is
        ↪ finished
        isDoingAction = false;
    }
```

- The entity must use an *idle* variable to know if a new action can be performed.
- Finally, in *Update* it is checked if the system is free to execute a new action:

```
protected void Update () {
        base.Update();

        if (idle)
        {
            StartCoroutine(AIAction());
        }
    }
```

The normal enemies have deterministic behaviors, that is, the player can know when and how the enemies will attack depending of the situation. So every enemy implements its own mechanisms to set an action, including stopping actions at the middle of its execution (e.g. when the enemy dies).

On the contrary, most of the routines and attacks of the bosses are executed randomly, so every boss has an additional AI script to decide which action is to be performed.

For random actions we use a WeightedRoulette. Each action is assigned a weight, so some actions are executed more probably than others. This kind of behavior is the used by the final bosses:

Figure 3.50: Lightning boss action weights

Normal enemies

Every enemy prefab maintains a specific structure for every script. The minimum components are the next:

- At root prefab:
  - Rigidbody: It provides Physics properties to the enemy.
  - Collider: The player must be able to perform attacks over the collider to damage the enemy
  - Animator: Object that links the animation controller (Animation State Machine) and the avatar (the bone armature)
  - Enemy logic component: The behavior to move and attack
  - Blinker: When an enemy is hurt, its meshes fade to red.
  - Damage Display: Every time an enemy receives a hit, the quantity of damage is displayed
  - Health Canvas Bar: A custom middleware script to manage the health bar behavior based on the events received from EnemyCharacter.
- Inside prefab:
  - Mesh: It contains the mockup and the real model, including all the bones and required guns and DamageArea objects to attack.
  - Health Canvas prefab: The health bar displayed over the enemy when it is damaged.

The Blinker component was implemented to show if the player attacks were really colliding against the enemy collider. For example, melee weapons only perform damage at specific times of the animation, so if an enemy enters before the collision check, it will not be affected.

Damage Display and Health Canvas Bar are adapted from Health Script by YounGen Tech[62]. Every time an enemy is damaged these components receive the event and update their data.

The next components aren't strictly necessary, but most of the enemies include them to implement their behavior. All of them are inside the prefab:

- AI: The AI Rig component from the RAIN AI module.
- Damage areas and Damage stats: To perform attacks.
- Guns: Custom implementations of *ThrowMagic* class to shot projectiles.
- Additional meshes and particle systems.

Once the enemy is implemented, the root must be moved near the floor because when an enemy is spawned, it appears at a specified height. So, all the meshes, colliders and damage area must be moved up.

AI

The implementation of the enemies' behavior is a combination of the RAIN AI navigation capabilities, the custom AI implementation and the specific actions for each one. An enemy family shares all the actions available using an abstract class, each variant uses the necessary actions based on its purpose.

Every enemy has its own particularities and special behaviors. It is decided to explain the behavior of Enemy Lightbomb Explode 3.51 which it has attacks that can be found in most of the enemies.

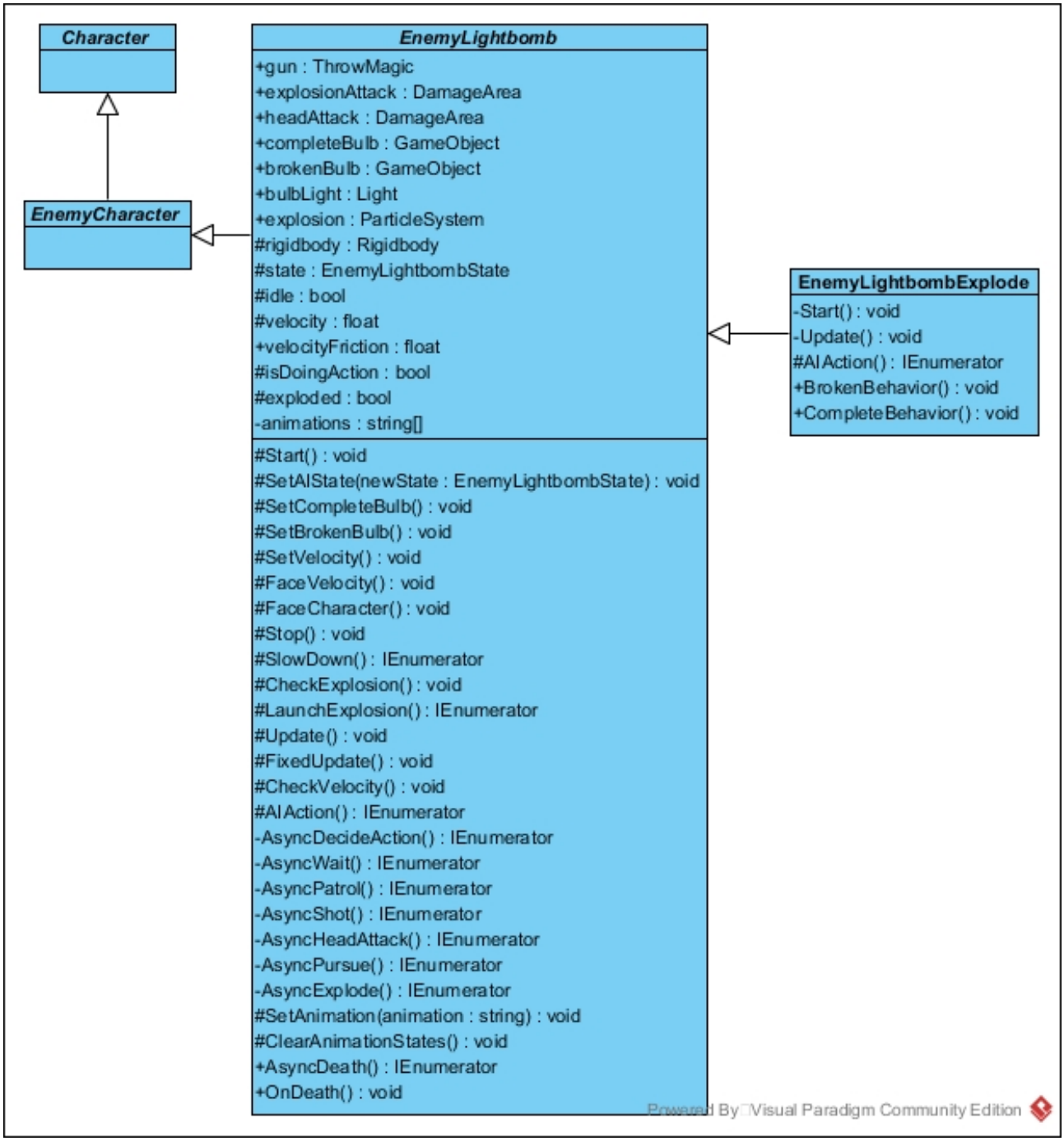


Figure 3.51: EnemyLightbombExplode and EnemyLightningbomb class diagram

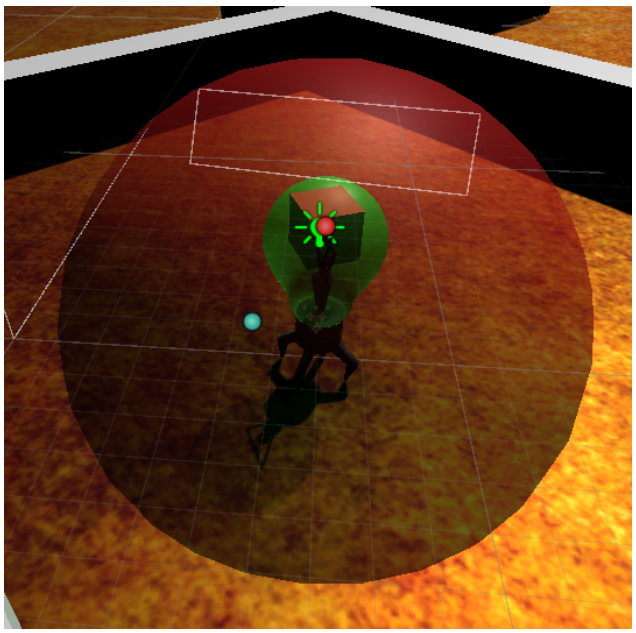


Figure 3.52: Enemy Lightbomb Explode with debug data activated

First of all it is presented the special elements that compose this enemy. Figure 3.52 shows the view of the Unity editor more additional graphics to debug:

- SphereDamageArea: The big red sphere. When the explosion attack is performed, it is checked if player is inside this.
- BoxDamageArea: The square inside bulb. It is used to check if player is touching it when the head attack occurs.
- Explosion particle system: The "shurikens" icon in the middle of the enemy. It is activated when the explosion behavior is executed.
- Gun: The green dot. A ThrowMagic implementation similar to the one that has MainCharacter. It Shots green projectiles.
- Bulb: There are 4 meshes, complete and broken, with inner and outer face versions of the same mesh.
- Mesh: All the visible mesh except the bulb. It has an AnimationController to animate the enemy depending of the action.

The AI implementation interacts with these elements as follows.

In the initialization the *RAIN AI* movement starts, the first AI state is waiting (between 1 and 2 seconds), and it is set the complete bulb:

```
void Start () {
  base.Start();
  tRig.AI.WorkingMemory.SetItem<bool>("move", true);
  SetAIState(EnemyLightbombState.WAIT);
    SetCompleteBulb();
}
```

Update function executes the default behavior of the classes that it extends (*EnemyLightbomb*, *EnemyCharacter* and *Character*). If the enemy dies, then it stay still. If enemy is alive and does nothing, it is allowed to move using *RAIN AI*.

The health of the enemy is checked at each frame (CheckExplosion()). If health is less than 50% the explosion routine is launched.

```
void Update () {
    base.Update();

    CheckExplosion();

    if (isDoingAction || !isAlive)
    {
        Stop();
    } else if (currentVelocity == 0)
    {
        tRig.AI.WorkingMemory.SetItem<bool>("move", true);
    }
}
```

AsyncExplode executes the animation for the bulb explosion and waits some time to perform the animation. Then it swaps the complete bulb to the broken one, activates the particle system with the visual explosion and checks if the player is inside the explosion collider. Finally, it waits for the animation to end:

```
protected IEnumerator AsyncExplode() {
    isDoingAction = true;
    ClearAnimationStates();
    animator.SetTrigger("bulb_attack");
    yield return new WaitForSeconds(0.3f);
    SetBrokenBulb();
    explosion.Play();
    explosionAttack.LaunchAttackCollisions();
    yield return new WaitForSeconds(0.1f);
    isDoingAction = false;
}
```

All the actions are invoked using coroutines to execute a set of actions with specific timing. Depending on the enemy state, it executes different functions.

```
protected override IEnumerator AIAction() {
    idle = false;
    yield return StartCoroutine(AsyncDecideAction());
    idle = true;

    if(exploded)
    {
        BrokenBehavior();
    } else
    {
        CompleteBehavior();
    }
}
```

If the explosion has occurred, the enemy try to reach the user and if distance is less than 3.5 distance units, the head attack routine is executed.

```
void BrokenBehavior() {
    SetAIState(EnemyLightbombState.IDLE);
    if (!isDoingAction && Vector3.Distance(this.transform.position,
    ↪ mainCharacter.transform.position) < 3.5f)
    {
        SetAIState(EnemyLightbombState.HEAD_ATTACK);
    }
}
```

AsyncHeadAttack stops the enemy, executes the animation and waits some time until the animation reaches the correct position. Next, the *BoxDamageArea* in its head checks if the player is colliding with it, doing damage to the player if this happens. Finally, more time is waited until the animation is finished.

```
IEnumerator AsyncHeadAttack() {
    isDoingAction = true;
    ClearAnimationStates();
    SetAIState(EnemyLightbombState.HEAD_ATTACK);
    tRig.AI.WorkingMemory.SetItem<bool>("move", false);
    animator.SetTrigger("broken_attack");
    yield return new WaitForSeconds(0.25f);
    headAttack.LaunchAttackCollisions();
    yield return new WaitForSeconds(0.3f);
        isDoingAction = false;
    SetAIState(EnemyLightbombState.IDLE);
}
```

If the bulb is complete, the enemy will swap between two states: shot, with a similar procedures than head attack, and wait, where the enemy can move using *RAIN AI*.

```
void CompleteBehavior() {
    switch (state)
    {
        case EnemyLightbombState.SHOT:
            SetAIState(EnemyLightbombState.WAIT);
            break;
        case EnemyLightbombState.WAIT:
            SetAIState(EnemyLightbombState.SHOT);
            break;
    }
}
```

3.5.7 Weapons

The weapons the main character can use are interchangeable, even in runtime or in debug mode. For this reason a generic system was created.

Figure 3.53 shows how the character is displayed in the editor. The red figures correspond to melee weapon, and blue figures are related to magic, being the circles the placeholders of the hands to attach the weapons, the red diamond the center of the damage sphere (the white sphere is the area of effect) for melee weapons and the blue diamond the initial position to cast magic. The dash box inside the character mesh and a gray circle in the floor representing the RAIN AI Target for the enemies.

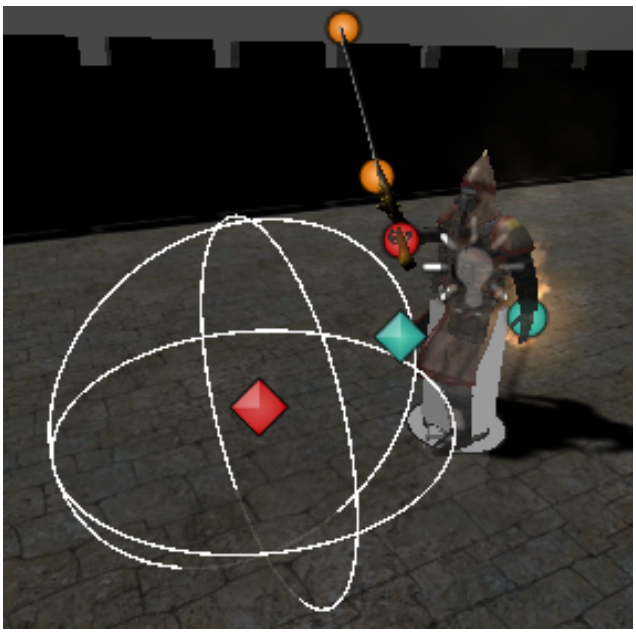


Figure 3.53: Main character gizmos

Implementation

When a weapon is replaced, It first deletes any previous weapon. Then, the new weapon is loaded from *Resources* folder. In the case of magic, all the spells must be in "Prefabs/Weapons/Magic/"+<name of the spell >+"/"+ <name of the spell >+"Prefab", so all the weapons must follow this name convention. For melee weapons we use the name "Melee". Once the prefab is instantiated, the weapon is initialized and an update of the stats is requested to assign the damage to the weapon:

```
1  void SetThrowMagic(string magicName)
2      {
3          if (throwEquipableMagic != null)
4          {
5              Destroy(throwEquipableMagic.gameObject);
6          }
7          GameObject newMeleeWeaponObject =
           ↪  Instantiate(Resources.Load("Prefabs/Weapons/Magic/" + magicName + "/" +
           ↪  magicName + "Prefab", typeof(GameObject))) as GameObject;
8          throwEquipableMagic =
           ↪  newMeleeWeaponObject.GetComponent<ThrowEquipableMagic>();
9          throwEquipableMagic.Init(this, magicWeaponPlaceholder.transform, magicName);
10         mainCharacterController.throwMagic = throwEquipableMagic;
11         effectsController.TriggerEffectsChanged();
12         Debug.Log("Set magic weapon: " + magicName);
13     }
```

Melee

Every melee weapon has the following structure:

- Melee Weapon Trail: Manages the colorful trail displayed during the attack.
- A subclass of *MeleeAttack*: The custom implementation for this weapon.
- Melee Damage Stats: The damage this weapon causes, with the addition of the damage multiplier managed by *MainCharacter*.
- Audio Source: The sound played during an attack.
  - Weapon mesh: The design of the weapon
    - start: An empty Transform that indicates the starting position of the blade (orange circle).
    - end: An empty Transform that indicates the final position of the blade (orange circle).

When the player executes an attack, the animation trigger for melee attacks is activated, playing the animation configured for the current weapon. At some moments of the animation, the function *CheckAttackCollisions()* is called, but it is finally the melee weapon who performs the attack according to its own implementation. In Figure 3.54 we can be seen this attack has three subattacks.

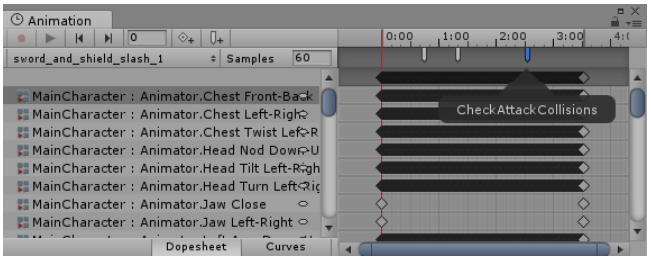


Figure 3.54: Melee attack animation

Magic

The combat system for magic spells works exactly like a melee weapon but with some special considerations:

- Instead of a physical weapon, a magic spell is represented by a colored flame.
- Every spell requires a more specific implementation.
- When a spell is cast, the magic spell transfers the damage and the owner to the projectile. Once launched, it becomes independent of the spell.
- All the spells use the same animation but they internally work as independent animations.
- The projectile collisions must be configured to collide against enemies, walls and obstacles.

For magic, the projectiles causes the damage, so no *DamageArea* is used. The projectiles contain all the logic to detect collisions with enemies.

3.5.8 Environment

Level generation

The normal levels of the games are generated using a custom version of the *Deluneay Triangulation* implemented by Nathan Williams. At the start-up of the level, it is generated a logical representation of the map (Figure 3.55). A set of quads is spawned at the same point, the first objective is to

displace these quads to avoid collisions. Once this condition is reached, the *Deluneay Triangulation is performed*, indicating which rooms must be connected.

In the last step, the final appearance displayed in the game is created using a room connector and complex system to decide which type of wall fits a tile depending on the surround tiles (Figure 3.56).

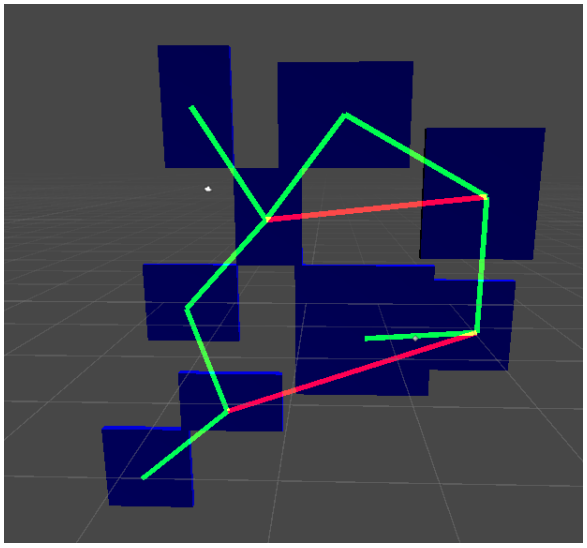


Figure 3.55: Logic map

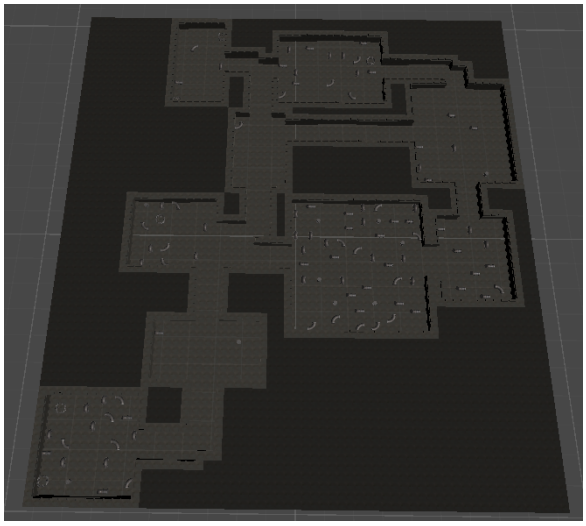


Figure 3.56: Final map

Room logic

Every room in a level contains a component called *DungeonRoom* to manage its behavior. At start-up, the obstacles are generated and the barriers rise. The navigation map creates a mesh for the movement of the enemies. Then, the barriers fall during the level fade and the smoke particle system is spawned, filling the room. When the enemies are spawned, *DungeonRoom* subscribe to the death event of these and keeping tracking of the number of enemies alive. Once an enemy dies, the component checks to see if it is the last one. If it happens, an event with the room is sent to *GameState* to decide if a portal must be spawned.

3.5.9 Internal logic

In this section they are explained some independent parts not related with the rest of the sections, but important to obtain a better vision about how the game works.

Levels modelization

LevelsInfo is the manager that decides the evolution of the game. Specifically, the levels and bosses the player must cope with to end the game. Every time the player starts a game, *LevelsInfo* is configured. It is *serializable* because it must be stored using the save data system. A *LevelsInfo* instance is composed of the number of finished levels, the initial seed, the current seed and the list of worlds. The seeds and number of finished levels are used to reconstruct a run at a certain point. Every world is composed of a list of levels. The worlds and levels for each world are configured in the *Init function*. For example:

```
worldList = new WorldList();
World world1 = worldList.AddWorld();
world1.AddLevel();
world1.AddLevel();
world1.AddBossLevel("Tutorial", "Training Train");
```

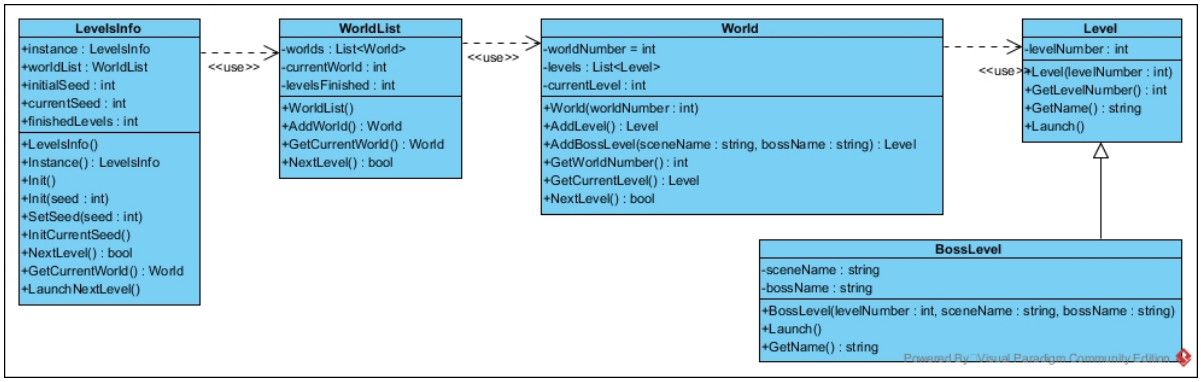


Figure 3.57: LevelsInfo class diagram

Save system

Unity provides the attribute *Serializable* for automated saving and loading of data using binary files. All the classes that require storage must use this attribute. As a disadvantage, they can't extend from *MonoBehavior*.

The game uses two types of data storage:

- Static storage in memory: When player goes to another level, all his information must be stored, that is: player statistics, current weapons, accumulated data of the run and effect bar values.

- Hard disk: When player ends a level, the game offers to save the progress. This data is written on a special folder of the computer managed by Unity, in Windows it is located in <User's folder>\AppData\LocalLow\ArtifactGear\MechanicalDystopia. The stored data is: user preferences (music, camera), the static storage in memory and the gifs showing the game over.

WeightedRoulette

During the development a recurrent issue occurs, due to the random nature of the game, lots of calls to the class *Random* provided by Unity will be used, but the probability to obtain a value is always the same. Most of the randomized content uses a custom implementation of a weighted roulette. This roulette allows obtaining random elements of an array but with configured probabilities, called weights. Weighted roulette is used in:

- Health items spawn
- Enemy spawn proportion
- Final bosses heuristics
- Portals and essences spawn probabilities

For example, if there are three items with weights: 8, 10 and 14; the sum is 32, so they have a probability of 8/32 (25%), 10/32 (31.25%) and 14/32 (43.75%).

World containers

In *Unity*, the resources found in the *Resources* folder can be accessed by path. Among other prefabs, there are three types of prefabs used to store which enemies, music and size of the level has each world. They are used as a static database. For example, when *EnemyManager* loads its data, it uses the next code to obtain the enemies. Notice the resource is a composed string:

```
1 int currentWorldLevel = LevelsInfo.Instance().GetCurrentWorld().GetWorldNumber();
2 enemyContainer = Resources.Load("Prefabs/WorldEnemies/WorldEnemies" +
  ↪  currentWorldLevel, typeof(EnemyWorldContainer)) as EnemyWorldContainer;
```

Item statistics

The artifacts and weapons displayed in workshop have statistics, available in annexes B. This data is generated using Excel and exported to CSV. At the game startup the *ObjectsManager* class reads the

three files corresponding to artifacts, melee and magic weapons.

The rules used to set the values are the next.

- Every item has a tier, from 1 to 5. For each tier it is configured an interval. The values for the eight statistic upgrades are selected randomly using the corresponding interval.
- The reduction value is the average of the eight upgrade values.
- The price is calculated as $price = reduction * 450 * (1 + random(-20, 20)$.

Similarly, the statistics for weapons are decided using simple but balanced formulae.

Health items

In the *MainCharacter* object, the *HealthManager* is founds. When a level starts, the probability to obtain health is 0%, and it is increased depending on the current health following rule:

$$currentPercentage = lastPercentage + \frac{1 - \frac{mainCharacter.health}{mainCharacter.maxHealth}}{100}$$

The system is implemented using the *WeightedRoulette*. Once player touches a health item, the item calls the function *ReceiveHealth(HealthObject healthObject)* from *MainCharacter*, the percentage of recovery is obtained and the health is increased based on this. The probability to obtain one type of health or another depends on the percentage of health of the player:

Player status	Small health weight	Medium health weight	Big health weight
health <30%	100	500	300
30% <= health <60%	300	500	100
60% <= health	500	100	1

Table 3.9: Weighted values for health item spawn probabilities

3.5.10 UI

Unity includes a powerful UI system, it was introduced in the version 4.6 to replace the previous UI toolset. With this system, all the UI can be developed using drag & drop and code. In general, the menus are created using a canvas and adding panels, buttons and texts.

For the gameplay cameras, the resolution is not an issue, but for UI, it matters. Every UI element adapts its size depending of the resolution, including the size of the text. We tested that in normal wide resolutions (720, 768 and 1080) the UI is displayed properly, maintaining the proportions.

Buttons

The UI buttons were created using the 9-sliced image type. From an image that includes the 9 sections of a button (Figure 3.58), Unity is able to compose the final layout independently of the size of the button (Figure 3.59).

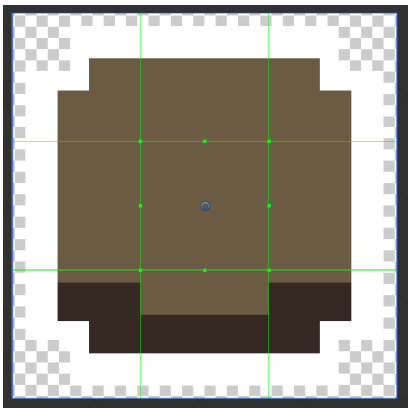


Figure 3.58: Button texture

Figure 3.59: Start button

Translation system

By default, *Unity* doesn't provide any method to display resources in other languages. The *Unity Asset Store* provides some packages to solve this issue. After testing some of the free assets finally it is decided to use *L20*, a translation manager for texts, images, audios and meshes. The translations must be declared in XML files, so that the addition of new language is a trivial process.

Every translation contains the id-translation pair for each line, for example:

```
<finished_rooms "Finished rooms">
<total_time "Game time">
<saving "Saving... please, don't close the game">
<switch_controls_keyboard "Keyboard and mouse">
<switch_controls_gamepad "Gamepad">
```

Every scene that requires translation must have the object *L20Settings*, which manages and updates the data when it is required. A translatable text requires the *L20n UI Text* component with at least the Identifier field filled. The text displayed is the identifier between diamonds. It is the task of the L20Settings to put the correct translation when the game is running.

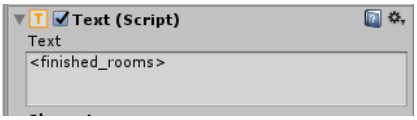


Figure 3.60: UnityEngine.UI. Text component

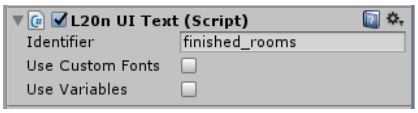


Figure 3.61: L20n UI Text component

Figure 3.62: Translated text

Finally, the translations can be configured using variables as a string formatter in a custom pseudo-programming language. For example, considering the number of elements, the gender, setting a translation or another depending on a specific variable.

Main menu

Figure 3.63: Main menu

The main menu (Figure 3.63) is presented as a set of interactive gears the user can click using its own gear cursor. When the cursor is properly set it starts to spin, making the central big gear also spin, and appearing the corresponding menu panel from one of the sides of the screen. Actually the first scene is another scene called *Start* that simply loads the *MainMenu* scene. This is done as a good practice (we can configure anything that must be done before the real game starts) and also due to the gear's materials have problems to be displayed correctly if it is the first scene. This is the appearance of the main menu in the editor:

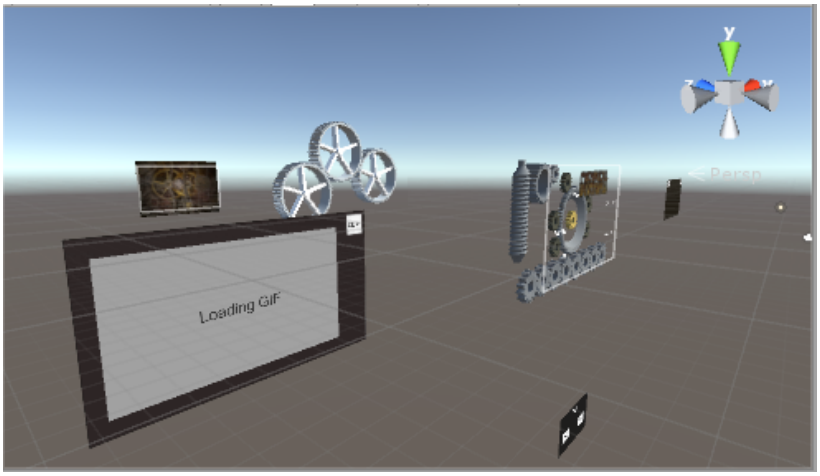


Figure 3.64: Main menu in Unity editor

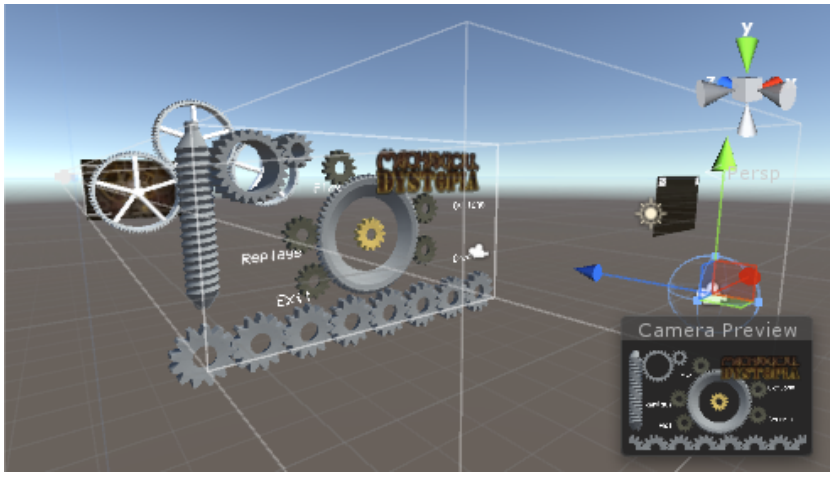


Figure 3.65: Main menu perspective

There are two cameras. The background camera displays the background image, some gears and a quad with a *SimpleGrabPassBlur* shader that blurs the objects beyond it.

Figure 3.66: Main menu background

The main camera renders the near elements, that is, the decorative gears, the title, the interactive gears and the UI texts.

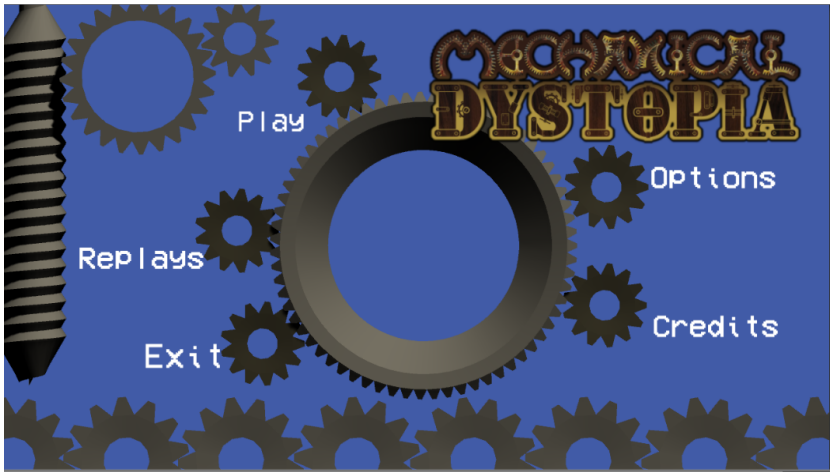


Figure 3.67: Main menu UI camera

The two cameras are necessary for the blurry effect because the gears' material provokes graphical errors on them. The gears and the worm are created using the paid asset *Procedural Gear*. This asset also implements realistic angular velocities for chained gears. All the panels in the main menu are in the scene, but out of the screen, they fade in and out using animations. Only the credits menu is loaded at runtime.

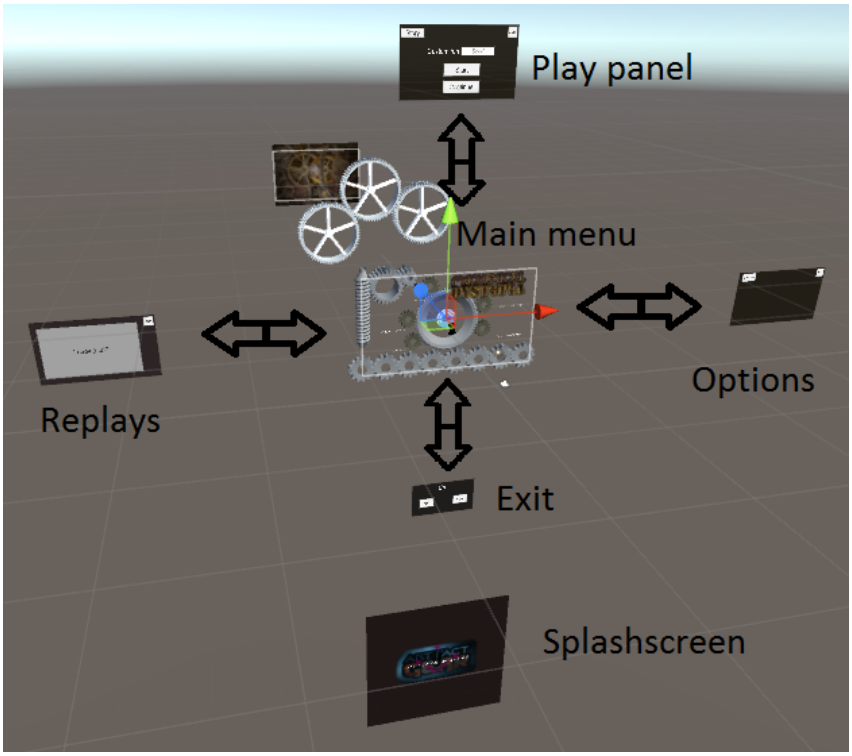


Figure 3.68: Main menu transitions

MainMenuLogic loads the stored data, if it exists, or creates the save slot if not. It also decides if the continue button of the Play panel is displayed (only if there is a saved run).

GearCursor uses *Raycast Phisics*, the yellow gear acts as the cursor of the player. If the player clicks over an element of the UI layer (all the grey gears around the big gear), the *GearCursor* calls the function *Launch* from the PanelLauncher component. The *FreeMovement* function unlocks the gear if it is connected to one of the grey gears.

PanelLauncher is in the grey interactive gears and launches the animation to fade in the corresponding UIPanel.

Every UIPanel has an animation state machine (Figure 3.69) with four states. The first time the trigger Fade is set, the *UIPanel* fades in to the screen, and when it is triggered again the *UIPanel* fades out.

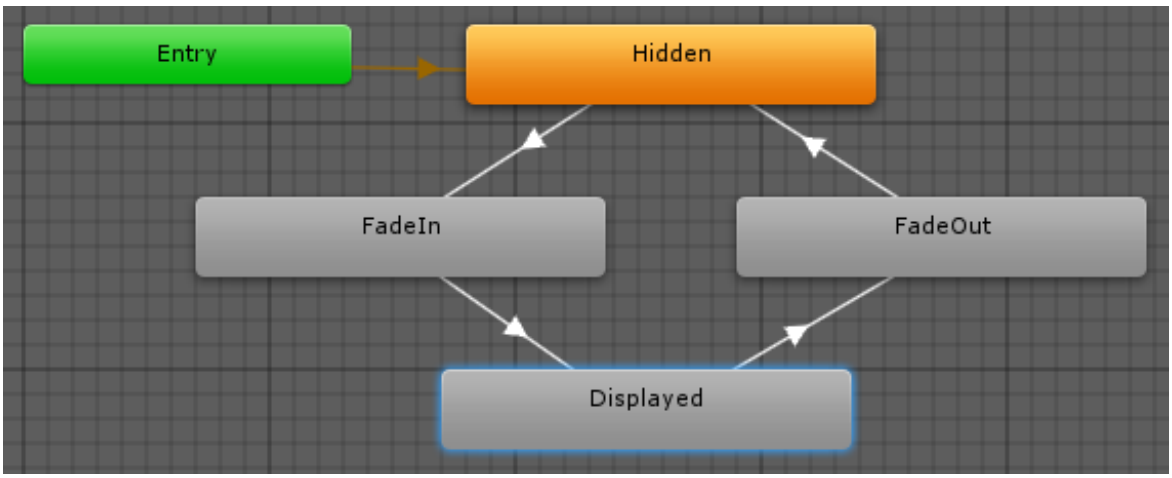


Figure 3.69: Main menu UI panel state machine

When the close button of a *UIPanel* is clicked, the Fade function of the *UIPanel* is called. At the end of the *FadeOut* animation the *ReleaseGearCursor* function is called.

Credits

None of the packages in the Asset Store convince me, so I decided to create a new implementation for the credits screen. Credits component has the configuration of the start position, separation between elements, types of elements and other necessary components to work correctly. When the credits are to be showed, a coroutine loads all the credits at runtime, filling the *Content* object with UI Text at the corresponding place considering the configured height and separation of each element.

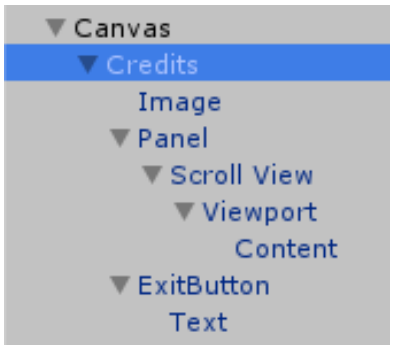


Figure 3.70: Credits structure

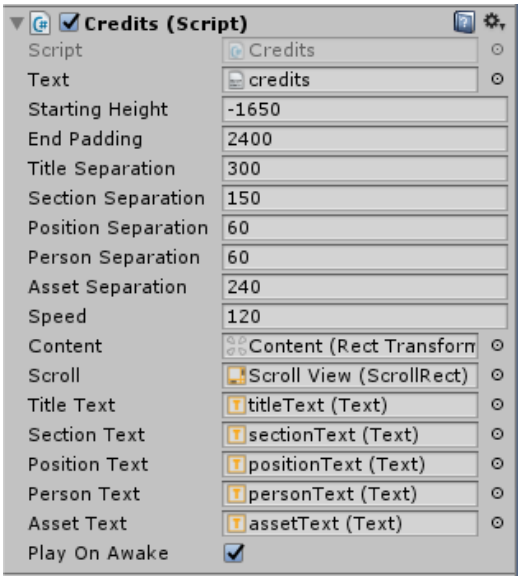


Figure 3.71: Credits component

This is an example of the XML file containing the lines displayed in the credits:

```
<?xml version="1.0" encoding="utf-8" ?>
<credits>
        <title>Mechanical Dystopia</title>

        <section>artifact_gear</section>

        <position>project_lead</position>
        <person>Sergi Vila Almenara</person>
        <person></person>

        <position>programming</position>
        <person>Sergi Vila Almenara</person>
        <person></person>

        <position>design</position>
        <person>Sergi Vila Almenara</person>
        <person></person>


        <section>assets</section>
        <asset>
                <name>Procedural Gear</name>
                <author>Proroc</author>
                <source>https://goo.gl/HYWIhG</source>
        <section>Music</section>
        <asset>
                <name>Dubmood - Chiptune</name>
                <author>Dubmood</author>
                <source>https://goo.gl/PxI25Q</source>
        </asset>
```

The available types are: title, section, position, person and asset. All of them are UI Text but the asset that is composed of two UI Texts and a button that opens the link in the default Internet browser of the system. Also, the types section and position uses *L20n* ids, so these texts will be translated at runtime.

On each frame the position of the content object is modified using the selected speed multiplied by *DeltaTime*. Since it is managed by a Scroll View, a smooth scroll effect is obtained, being movable with the mouse.

The *Credits* prefab is used in the Main Menu and also at the end of the game in the Credits scene. The use of the prefab in both scenes required a special configuration of the *RectTransforms* of *Panel*, *Scroll View*, *Viewport* and *Content* because there were some graphical errors about screen position and scale.

Book menu

To present the story and controls of the game we use the free version of the *Unity3D Book Page Curl* asset. It provides the logic about a 2D book whose pages can be turned with the mouse or buttons, being the set up of the pages a task for the programmer. The pages must be textured, so that all the text displayed is part of the image.

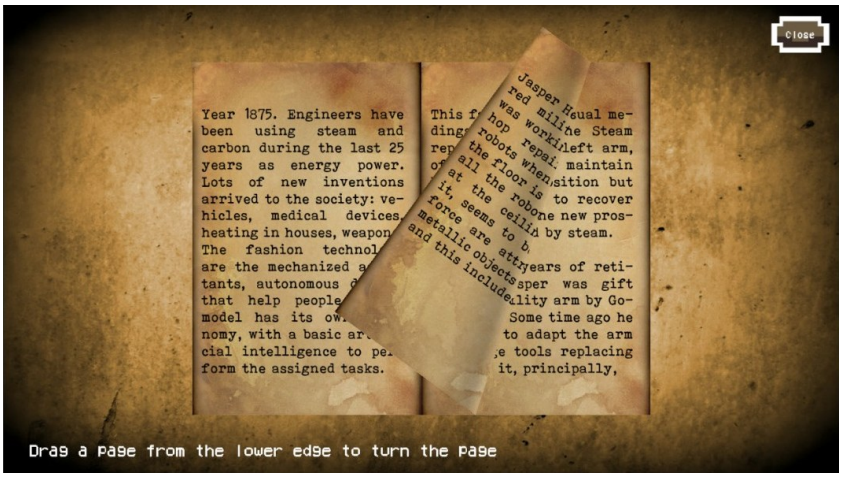


Figure 3.72: Story menu

The first time the user starts the game, the *Story Button* is the only available option in the Play menu. Once the user clicks on it, the Story Book is displayed. Then the Story page is flipped automatically and some pages of text are showed. By clicking on the bottom-right corner the current page can be

turned. After all the pages have been displayed, the Controls Button appears and the user can see the keys to execute actions in the game.

Figure 3.73: Controls menu

Finally, when the user presses the *Start game button*, the game starts as usual.

Chapter 4

Art

4.1 Introduction

A motivation for start the art of the game was given by the series of videos *From Programmer to Artist*[71] by Nathanael Weiss, a game developer that started as a programmer and finally became a one-army man, developing all the aspects (coding, art, music) of a game in his videogame company, Wizardfu.
He first recommends to break the psychological barrier that a programmer can't be an artist due to his methodical and technical thinking. He explains our first works don't need to be perfect. It is enough if it transmits the proposed ideas. You have to be clear on the searched visual appearance and style. He recommends to search in social networks for people sharing their work, and to do tutorials or timelapses. Regarding tools, he considers the use of a graphic tablet is essential. The mouse is not precise enought for 2D art. After that, a set of basic tips about lights, shadows, colors, shading and animations are explained about Photoshop. The information is simple but it's a beginning for newbie artists.

In the course *Videogame design for high-performance platforms* some objects of our project were to be done using Blender, so I had a bit of experience using it. But it was not enough to design all the elements of this game, including rigged and animated enemies, weapons, decoration and the main character, with some texturization.

The next sections explain how are created and obtained the resources for Jasper, weapons and final bosses, environment. Also some information about the use of protected content.

4.2 Weapons

The melee weapons the player can carry were created using Blender with reference images as templates (Figure 4.1). Every part of a weapon is usually an independent object. The next list presents how each part has been done:

- Handle: The start point of a weapon is the bottom part, normally a circular tube with little length variations. After setting the cylinder, we extrude and scale it following the shape of the template (Figure 4.2). Alternatively, if the handle is flat, it can be done taking a cube as starting point.
- Head: If it has a square shape, the method for head is used. Most of the heads are created starting with a single vertex, extruding it until all the shape is defined. The next step is to fill it with faces. Now there is a plane figure without volume. We can add the third dimension using the Solidify modifier.
- Edge: This part is made taking a plane, extruding it by the borders in each side, and adapting it with the correct width. If the edge is sharpen, a longitudinal loop subdivide is required (Figures 4.9 and 4.10). The last step is for providing the volume, using the mirror modifier with the clipping flag to snap the borders, or the solidify modifier if the surface is flat (Figures 4.11 and 4.12).
- Pipes: *Bézier* curves are the ideal technique for such shapes. Once the curve is done (Figure 4.3), resolution, fill and *Bever* options must be configured. Additionally if the pipe varies its width, another Bézier curve can be used as *Taper Object* (Figure 4.4).
- Details: Most of the screws and other little pieces are created using prisms, spheres and gears (Figures 4.5 and 4.6). More elaborated forms are created with *Bézier* curves (Figures 4.7 and 4.6).

Once all the weapon parts are completed, they must be joined to a single object for the texturization.

For texturizing the final model all the faces must be unwrapped using the *Project from View* being the view (front ortho) obtained pressing "1" numpad key (Figure 4.14). In the *UV/Image editor* there are the faces in the desired form. They must be scaled, rotated and moved to match the template (Figure 4.15).

At this point the model is finished (Figure 4.13). It only remains to export it to a .fbx file (Figure 4.16).

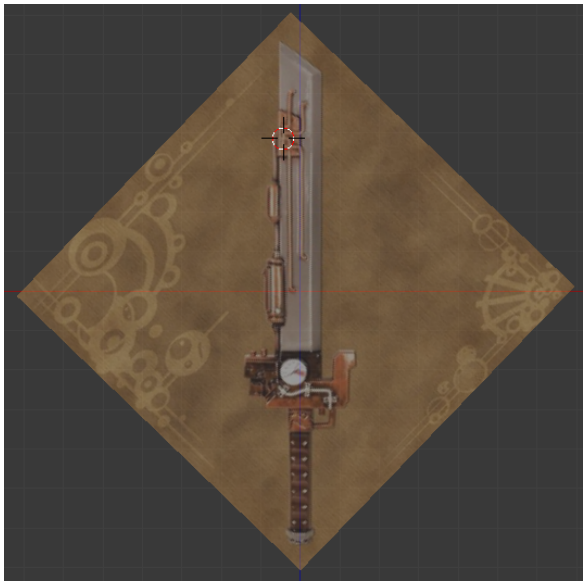


Figure 4.1: Weapon template

Figure 4.2: Weapon handle

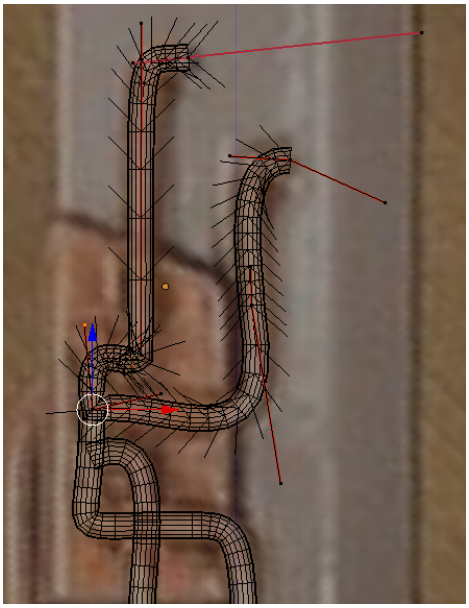


Figure 4.3: Pipes

Figure 4.4: Pipe with variable width

Figure 4.5: Basic weapon details - Wireframe

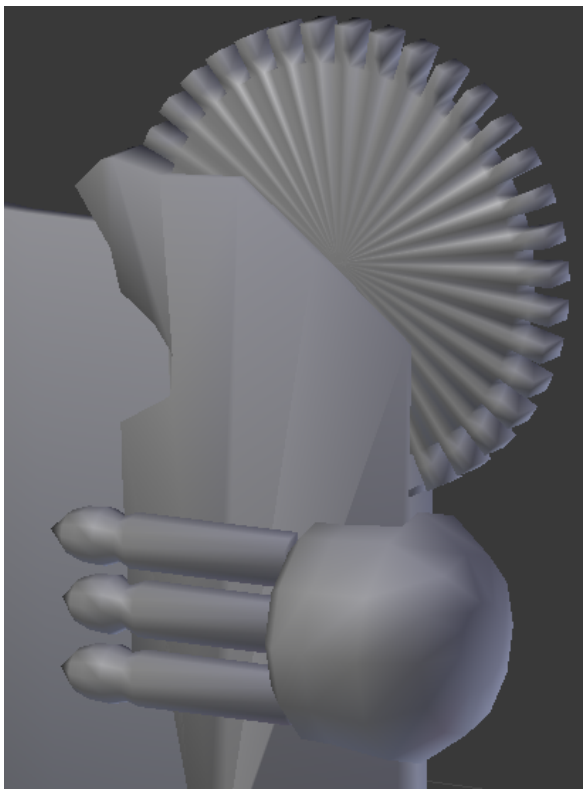


Figure 4.6: Basic weapon details - Solid

Figure 4.7: Weapon detail created with Bézier curve

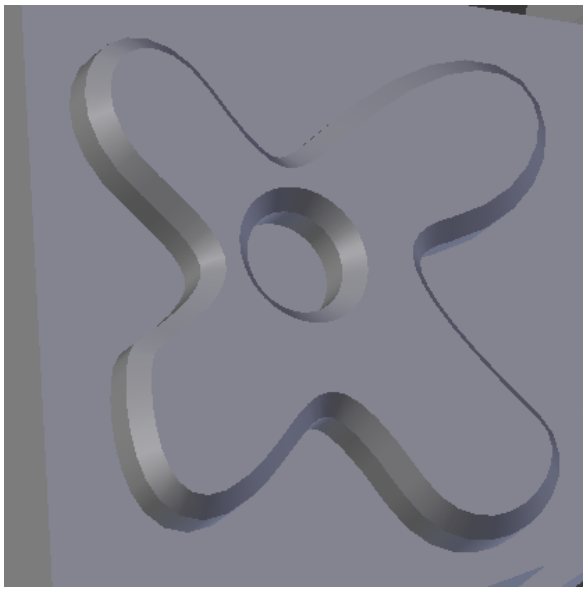


Figure 4.8: Weapon detail with Solidify modifier applied

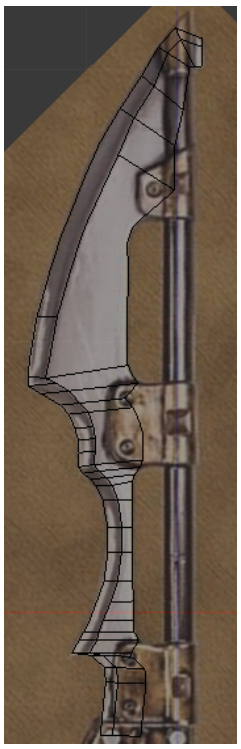


Figure 4.9: Flat edge wireframe

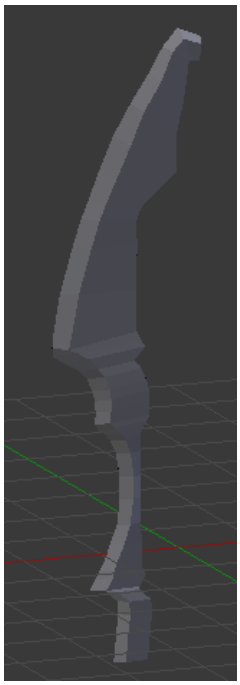


Figure 4.10: Flat edge solid

Figure 4.11: Flat edge

Figure 4.12: Flat edge with Solidify modifier applied

Figure 4.13: Finished weapon

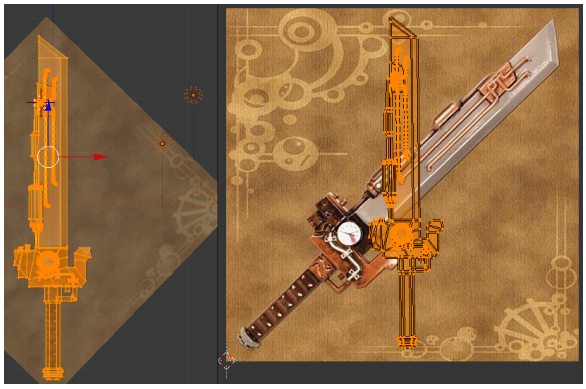


Figure 4.14: Faces unwrapped into UV mapping

Figure 4.15: Faces fit texture template

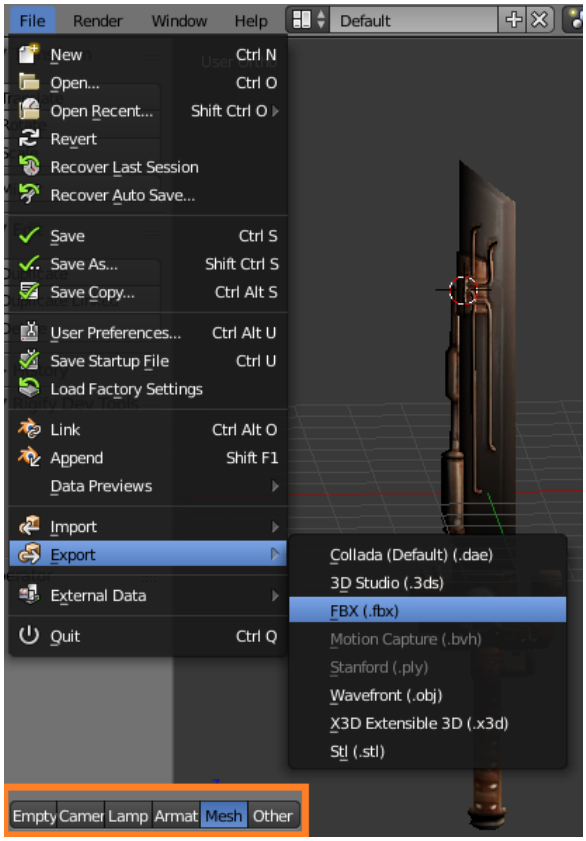


Figure 4.16: Exporting a weapon

4.3 Characters

Main character

During the development Jasper was the mage model (Figure 4.17) taken from *Warrior Pack Bundle 2 FREE*. The bundle includes animations for the character. Unity can manage humanoid characters and animations by itself, that is, any human animation is compatible with any human model. We could have be developed an original character. Some test were done, but the texture quality and resulting animations would be better. Finally, searching animations in *Mixamo*[61], we found Heraklios (Figure 4.18), a high detailed industrial character with lot of personality that fits with Jasper. By simply replacing the mage model the integration was done.

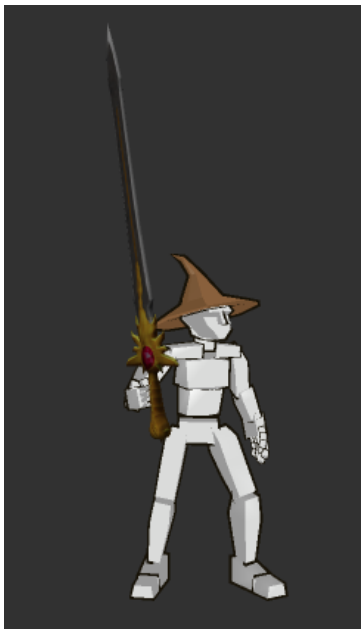


Figure 4.17: Main character prototype

Figure 4.18: Final main character

4.3.1 Enemies

Once most of the enemy routines were designed, they had to take shape. After starting the project, due to my few Blender skills, the use of the low-poly technique was the most recommendable, but after the design of the weapons and the firsts enemies, I discovered and learn a lot of features to obtain complex and detailed designs. The evolution design of the enemies took the following steps:

1. Basic description: Physical sketch and its role in the game.

2. Reference image: High-quality concept art done by an artist. Alternatively, if an image caused me a good impression, the original description was discarded or altered.

3. Blender design: If the reference image is a front view, the final model will be more reliable. Otherwise, some elements or shapes may vary due to some parts being covered by others.

4. Modifications: Some elements of the reference image will be simplified or made totally different.

5. Details: When the model is finished, more details and *steampunk stuff* are added.

All the models created are uploaded to my personal Sketchfab page: https://sketchfab.com/svila. In this webpage it can be seen 3D models in all detail.

4.3.2 Final bosses

Training Train

As part of the prototype, the model[73] for *Training Train* was obtained from *TurboSquid* with own textures. Figure (4.19) shows the original model, and Figure (4.20), the model integrated in the game.

Figure 4.19: Training Train original model

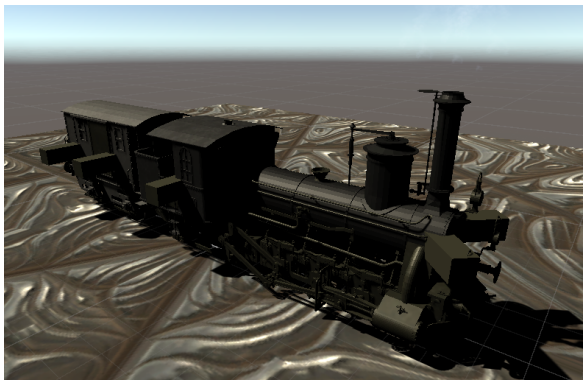


Figure 4.20: Training Train ingame model

Lightning Mask

The design of the boss mask is based on this picture:

Figure 4.21: Lightning Mask's mask

The most relevant Blender feature used on this model is the use of a *Lattice* modifier for the curvature of the mask and the nose section:

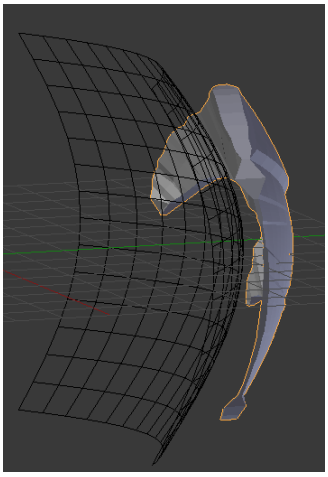


Figure 4.22: Lightning Mask curvature using Lattice modifier

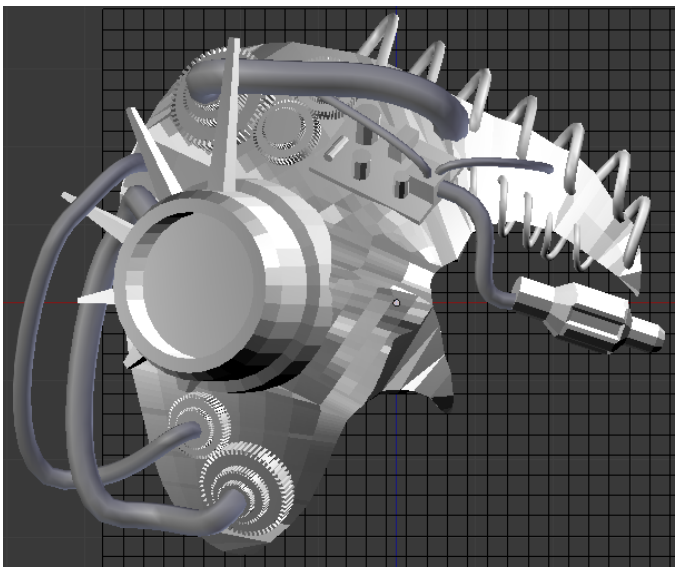


Figure 4.23: Lightning Mask finished mask

The body of the character is done using the plugin *Add chain*, Figure 4.24.

The aspect in the game is shown in Figure 4.25.

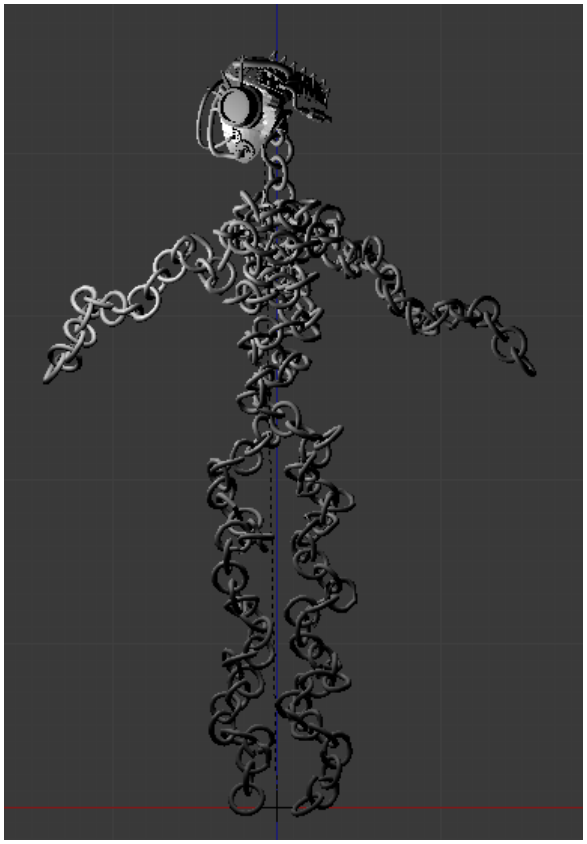


Figure 4.24: Mask entire body

Figure 4.25: Lightning Mask ingame design

mAlice

The base image for the body of the character is the next:

Figure 4.26: mAlice concept art

Being the finished body:

Figure 4.27: mAlice's body

The head was created following this tutorial [8]:

Figure 4.28: mAlice's head

Once both parts were completed they are joined in the same Blender project:

Figure 4.29: mAlice textured

- Use of seams: To select all the vertices of an object we used the L key. Seams are used to mark the bounds of a group of faces (Figure 4.30). It is useful to split cloths or parts of a a body for the unwrap process. In this case 3 seams were created to divide the body: neck, shoulders and waist. The remaining objects (for example the wings) can be selected individually since their vertices are not related to others.

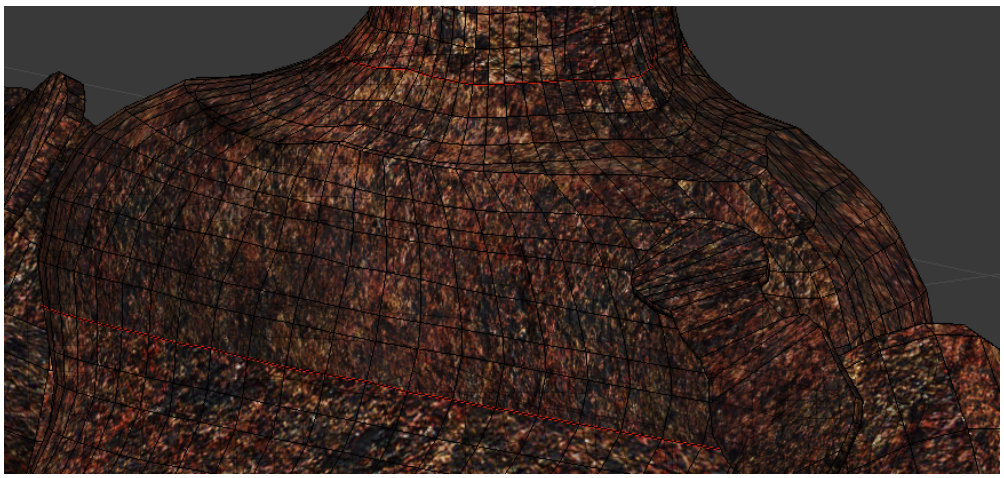


Figure 4.30: mAlice Use of seams

- Vertex group: Every part of the character is split into a set of vertex groups (Figure 4.31). If a part of the body is to be used frequently, this technique facilitates the selection of all the vertices quickly.

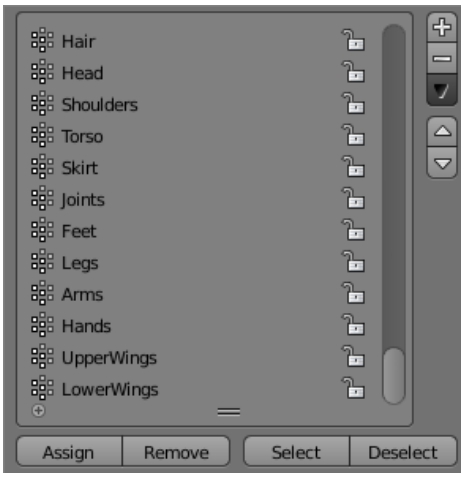


Figure 4.31: mAlice Vertex groups

- Partial Smart UV Project: Selecting every vertex group and applying a Smart UV Project we obtained a squared set of unwrapped faces. With this method every part can be textured easily. The final UV mapping is scaled to obtain better graphics maintaining the same resolution texture size:

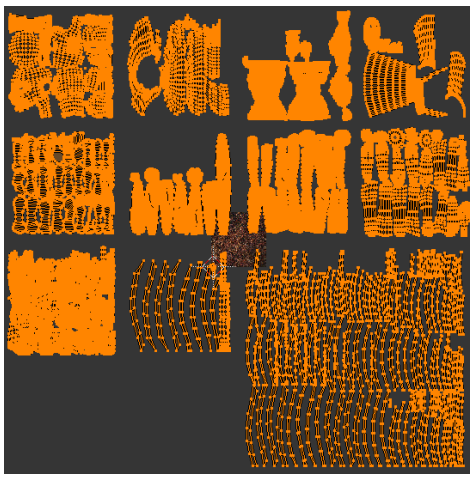


Figure 4.32: mAlice UV Smart Unwrap

The final aspect of mAlice is shown in Figure 4.33.

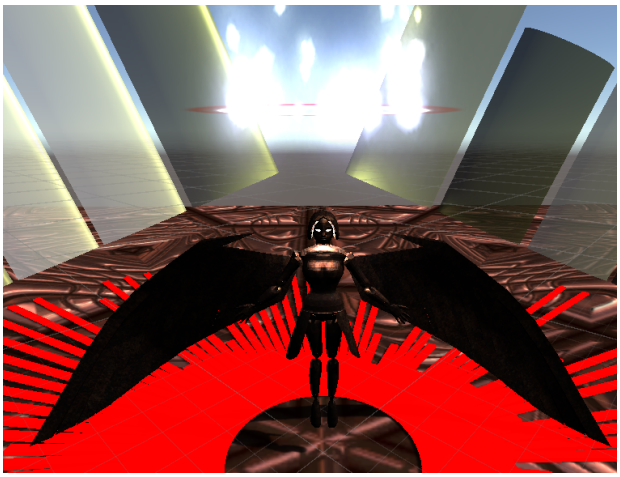


Figure 4.33: mAlice ingame design

The Creation

The inspiration for The Creation is showed in Figure 4.34. Due to its industrial aspect, most of the pieces are flat prisms. The entire model was made from independent pieces, that are finally joined with *Boolean* modifier. The final aspect is displayed in Figure 4.35.

Figure 4.34: The Creation concept art

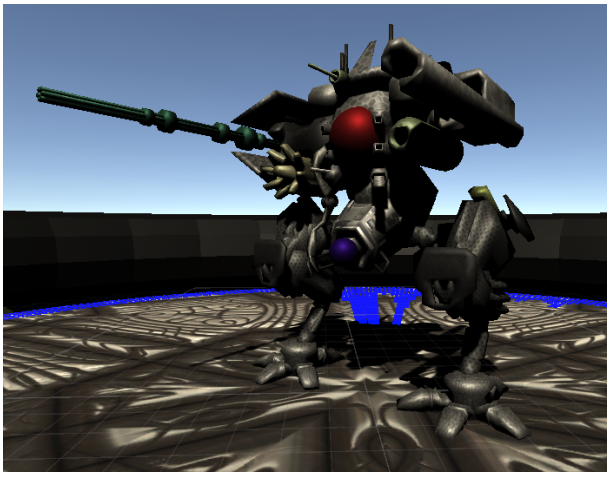


Figure 4.35: The Creation final design

4.4 Environment

The floor and walls of the levels are the default of the *Delaunay Triangulation* project. The obstacles and rooms assets were taken from *Pipes Kit* by mojo-structure. The portals are implemented using the assets from *Particle Collection SKJ 2016_Free samples*. We created a color configurator to obtain each color effect, but black colors were not possible, so black portal uses another asset, *TELEPORTER*. The fog of the unvisited rooms is the default black smoke of the Standard Assets. Every fog is contained inside a collision box that has the size of the room.

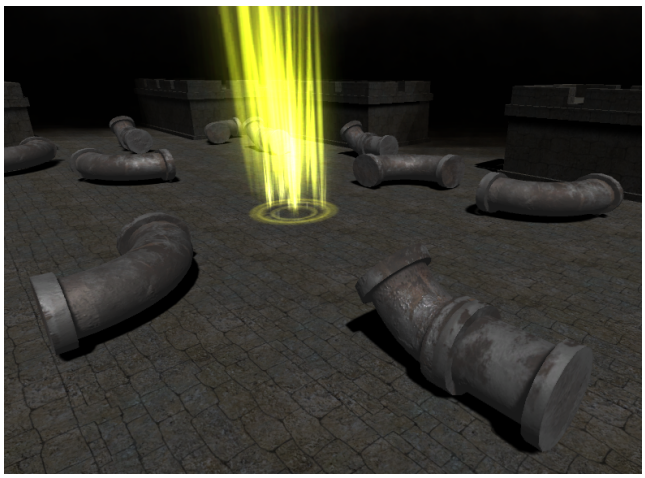


Figure 4.36: A room with a portal and obstacles

4.5 UI

Following the steampunk spirit, all the user interface is thematized with steampunk colors and gears. The images created with *Photoshop* for the game are cursor icon (Figure 4.37) and the logos for Artifact Gear (Figure 4.38) and for Mechanical Dystopia (Figure 4.39).

Figure 4.37: Gear cursor

Figure 4.38: Artifact Gear logo

Figure 4.39: Mechanical Dystopia logo

4.6 Music

The sound section of the game falls in the genre of chiptune and 8-bit. Every world has a selection of themes to avoid monotony. Also the final battles have their own theme. Most of the songs are obtained from YouTube and SoundCloud from authors that share their own work, and even urge people to use their songs in other projects. Annex D shows all the music used.

4.7 Copyright

We tried to use textures without copyright or with licenses that permit the alteration of the original work. In some cases this was not been possible, for example, the textures for the weapons, that comes from a shop that sells real plastic weapons.

Most of the used songs come from *YouTube* channels that create their own covers from original songs, that is, the rights of the songs are from the *YouTube* authors, not the original musicians. So I contacted with some of them to obtain their permission. The answer was affirmative in all the cases. Some of them explained to me that the game can't be publicly available.

The used resources and their authors are available in the credits of the game and in the annexes of this document. In this sense, every attempt has been made to take all the authors into consideration so that they receive the recognition they deserve.

For the issues explained, the availability of the game is very limited. First of all no remuneration can be obtained due to there are not known authors or licenses for some contents. It is the case of some songs that are being sold in digital stores.

To avoid any kind of legal problem, the game requires online authentication on the company's server. At any moment, the game can be deactivated an nobody can be able to play. The game checks the next file: http://artifactgear.com/validation.json

In conclusion, if all the used content with copyright issues is replaced with original own content or other resources with less restrictions, the game can be shared or sold without problems. This is not being fulfilled, the access to the project must be restricted because of the contents with copyright.

Chapter 5

Conclusions

5.1 Future Work

Currently the game is playable, it has specific game mechanics and enough content, but the truth is that it requires more time, resources and a consolidated company with specialized roles to offer a much better product.

The next points are some improvements that can be done with more time:

- Motion capture: The animations provided by Mixamo are of high quality, but their usage is for general purposes. The use of motion capture is probably the best technology to develop custom animations quickly.
- Minimap: It was thought the use of a map like Enter the Gungeon does, but finally this feature stayed in an idea. Using the assets *Hand Drawn & Real World Mini Maps* or *Minimap Script Package* it can be implemented in some hours.
- New effects: An implementation for temporal and a set of new effects for enemies were started. These are the names of the discarded effects: frost screen, multiscreen, noise, black hole, wiggle, headache, ripple, color blindness, thermal vision, negative and sobel.
- Upgrade Unity: The game is developed using the 5.4 version. At the time this document is being written, 5.6 version is available. Normally, when the version is upgraded, some code will be deprecated, scenes stop working, assets fail, so it is a frail task.
- Rework of graphics: Principally the scenarios must be recreated with more variety and details. Also some textures could be better and the inspiration for enemies and weapons is fine, but for

commercial project all the content should be original.

- New modes: Initially it was thought to add two additional modes. A random mode where the relation between stats and effects become random on each run, and easy mode, where effects and enemy damage are palliated to offer a best experience for newbie players. Also a daily/weekly fixed run mode was thought, as other roguelite do, in these modes everybody plays the same level and a ranking is displayed at the end, this feature can be implemented using DynamoDB from Amazon to download the active seeds and generate the rankings.
- Bugs: It's practically unavoidable to create a project of these magnitudes without unpredictable behaviors that provoke errors. Also, a Quality Assurance team must test the game to guarantee a good user experience.
- Statistics: Player stats, effects, levels, enemies and items have values that must be studied and balanced to obtain a fair and enjoyable game. This task could take months and thousands of hours playing the game. A statistical simulator should be created to obtain a perfect balance.

5.2 Conclusions

In this project a roguelite game has been implemented, following the same scheme as the ones that are great referent examples of the genre, but adding genuinity in its mechanics. Departing from the originally game design document, every aspect has been refined and because of a fair stage direction of the document we accomplished the proposed objectives, even adding new ones as the project was developing and new ideas were coming up.

Although we accomplished the objectives, not all the development of the project has been made in the times we scheduled at the beginning because of difficulties such as: mistakes of programming that were delaying other tasks, misadjusted assets according to our needs, contents that we finally decided not to use, builds that did not even finish or that were not working in the editor, adjustments in the models and animations so that were functional in Unity and excessive richness of details that is not seen in the isometric view. Even though we also learn from mistakes, these delays caused we could not do new tasks.

Because of Artifact Gear, the fictitious company that developed Mechanical Dystopia, has an only one member, this one has had to assume the following roles videogames designer, programmer, graphic artist, modeler, animator, screenwriter and translator. Being my main role a programmer, my art

skills have improved drastically, specially in the 3D design, being capable of modeling practically every element in a mid level.

I personally think that Mechanical Dystopia is a good start to develop major projects, but I learnt the lesson that the best way to develop a project is counting on the cooperation of a multidisciplinary team. The development of videogames requires great knowledge in many disciplines and much time. Moreover, working as a team, we have a variety of opinions and ideas that can enrich the results of the task. On the other hand, working alone gave me the total control of the project and I learnt many techniques. If this project had been developed by a team, in which every worker had had a defined role, the results would have been better and the contents would have been richer.

To sum up, this project has proved that people can develop a videogame if they show interest and devote time working on it, and it does not only depend on their skills.

Bibliography

[1] 2K. Bioshock. http://www.bioshockgame.com/.

[2] Activision. Crash bandicoot. http://www.crashbandicoot.com/.

[3] Activision. Guitar hero. https://www.guitarhero.com/.

[4] Alastair Aitchison. Animal crossing curved world shader. https://alastaira.wordpress.com/2013/10/25/animal-crossing-curved-world-shader/.

[5] Alastair Aitchison. Super mario galaxy "reveal hidden platforms" cg shader. https://alastaira.wordpress.com/2013/09/07/super-mario-galaxy-reveal-hidden-platforms-cg-shader/.

[6] Sergi Vila Almenara. Artifact gear. http://artifactgear.com/press/.

[7] Amazon. Amazon web services. https://aws.amazon.com/.

[8] Alimayo Arango. How to model a human female head in blender 2.74. https://youtu.be/dQlEWvVwqKo.

[9] Arduino. Arduino. https://www.arduino.cc/.

[10] Glenn Wichman Artificial Intelligence Design, Michael Toy. Rogue 1983. https://archive.org/details/msdos_Rogue_1983.

[11] Atarihq. Source. http://www.atarihq.com/tsr/special/tetrishist.html.

[12] BioWare. Mass effect. https://www.masseffect.com/.

[13] Half Burnt Biscuits. Source game engine: Fruit slicing game – template free. https://cgsourcegame.com/product/source-game-engine-fruit-slicing-template/.

[14] GoDaddy Operating Company. Godaddy. https://godaddy.com.

[15] Dbolical. Indiedb. http://www.indiedb.com/.

[16] Fira de Barcelona. Barcelona games world. http://www.barcelonagamesworld.com/.

[17] Doxygen. Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[18] Square Enix. Deus ex. https://www.deusex.com/.

[19] Inc. Enterbrain. Rpg maker. http://www.rpgmakerweb.com/.

[20] ephtracy. Magicavoxel. https://ephtracy.github.io/.

[21] Explosive. Warrior pack bundle 2 free. https://www.assetstore.unity3d.com/en/#!/content/42454.

[22] Josh Fairhurst. Think before you bundle. http://www.gamasutra.com/blogs/JoshFairhurst/20140710/220768/Think_Before_You_Bundle.php.

[23] FileZilla. Filezilla. https://filezilla-project.org/.

[24] Blender Foundation. Blender. https://www.blender.org/.

[25] Ubisoft GamersObservation. Rayman legends, all music levels. https://youtu.be/7m5YQrucis8?t=19m25s.

[26] Brace Yourself Games. Crypt of the necrodancer. http://necrodancer.com/.

[27] Cellar Door Games. Rogue legacy. http://www.cellardoorgames.com/roguelegacy/.

[28] Dabadu Games. It's full of sparks. http://dabadugames.com/press/sheet.php?p=its_full_of_sparks.

[29] Digital Sun Games. Moonlighter. http://moonlighterthegame.com/.

[30] Epic Games. Unreal engine 4. https://www.unrealengine.com/.

[31] Spearhead Games. Stories: The path of destinies. http://www.storiesthepathofdestinies.com/.

[32] Khronos Group. Opengl. https://www.opengl.org/.

[33] Joachim Holmér. Shader forge. http://www.acegikmo.com/shaderforge/.

[34] IndieGoGo Inc. Indiegogo. https://www.indiegogo.com/.

[35] Plunge Interactive. Plunge interactive. http://www.plungeinteractive.com/.

[36] Kickstarter. Kickstarter. https://www.kickstarter.com/.

[37] kode80. Pixelrender for unity3d. http://kode80.com/blog/2016/02/23/pixelrender-for-unity3d/index.html.

[38] Konami. Metal gear solid v. https://www.konami.com/mg/mgs5/.

[39] Petri Kuittinen. Rogue - exploring the dungeons of doom (1980). https://web.archive.org/web/20071217100401/http://users.tkk.fi/~eye/roguelike/rogue.html.

[40] Robot loves Kitty. Legend of dungeon. http://robotloveskitty.com/LoD/.

[41] Heart Machine. Hyper light drifter. http://www.heart-machine.com/.

[42] Altered Matter. Etherborn. http://alteredmatter.com/etherborn/.

[43] Edmund McMillen. Binding of isaac. http://bindingofisaac.com/.

[44] Kenneth C. R. C. Arnold Michael C. Toy. A guide to the dungeons of doom. https://docs.freebsd.org/44doc/usd/30.rogue/paper.pdf.

[45] Microsoft. Id@xbox. http://www.xbox.com/es-ES/developers/id.

[46] Namco. Pacman. http://pacman.com/.

[47] Bandai Namco. Dark souls. https://www.darksouls3.com/es/.

[48] The Ministry of Peculiar Occurrences. What is steampunk? http://www.ministryofpeculiaroccurrences.com/what-is-steampunk/.

[49] Vincent Paquin. Unity shader : Silhouette. http://vincentpaquin.blogspot.com.es/2013/02/unity-shader-silhouette.html.

[50] Dodge Roll. Enter the gungeon. http://dodgeroll.com/gungeon/.

[51] Sega. Binary domain. http://www.sega.com/games/binary-domain%E2%84%A2-0.

[52] Unity Technologies SF. Unity. https://unity3d.com.

[53] Stefan Sigl. Dissolve shader. https://www.youtube.com/watch?v=DceLSyaRnMA.

[54] Sony. Playstation talents. http://www.playstationtalents.es/.

[55] Steam. Steam early access. http://store.steampowered.com/earlyaccessfaq/.

[56] Steam. Steam greenlight. https://steamcommunity.com/greenlight/.

[57] Steam. Valve. http://store.steampowered.com/.

[58] 8-Bit Studio. Skara: The blade remains. https://playskara.com/es/.

[59] Arkane Studios. Dishonored. https://dishonored.bethesda.net/.

[60] SuperGiantGames. Bastion. https://www.supergiantgames.com/games/bastion/.

[61] Adobe Systems. Mixamo. https://www.mixamo.com/.

[62] YounGen Tech. Survival shooter tutorial. https://www.assetstore.unity3d.com/en/#!/content/7112.

[63] Unity Technologies. Survival shooter tutorial. https://www.assetstore.unity3d.com/en/#!/content/40756.

[64] Rival Theory. Rain ai. http://legacy.rivaltheory.com/rain/.

[65] GAME TROOPERS. Tiny troopers. https://www.microsoft.com/es-es/store/p/tiny-troopers/9wzdncrdljw7.

[66] GAME TROOPERS. Tiny troopers 2. https://www.microsoft.com/es-es/store/p/tiny-troopers-2-special-ops/9wzdncrdljwk.

[67] Anomalous Underdog. Melee weapon trail. https://www.assetstore.unity3d.com/en/#!/content/1728.

[68] Vlambeer. Nuclear throne. http://nuclearthrone.com/.

[69] Vlambeer. presskit(). http://dopresskit.com/.

[70] Vlambeer. Wasteland kings. https://vlambeer.itch.io/wasteland-kings.

[71] Nathanael Weiss. From programmer to artist. http://wizardfu.com/book/from-programmer-to-artist/.

[72] Nathanael Weiss. Wizard fu. https://wizardfu.com/.

[73] Locomotive works. Steyerdorf steam locomotive. https://www.turbosquid.com/3d-models/steam-locomotive-3d-model/782987.

[74] xPaw and Marlamin. Steamdb. https://steamdb.info/.

[75] YoYogames. Game maker. https://www.yoyogames.com/gamemaker.

[76] Derek Yu. The full spelunky on spelunky. http://makegames.tumblr.com/post/4061040007/the-full-spelunky-on-spelunky.

Appendix A

Roguelite videogames

Title	Launch year	Price (€)	Steam sales	Metacritic score	Developer	Publisher	Platforms	Engine	Genres	Thematic	Aesthetics	Wikis
Binding of Isaac	2011	4.99	2,920,839 - 3,011,029	84	Edmund McMillen, Florian Himsl	Edmund McMillen, Headup Games	Windows, Linux, Mac	Adobe Flash	Top-down, dual-stick	Dungeons, Chistendom, pop culture, horror	2D, cartoon	http://bindingofisaac.wikia.com/
Binding of Isaac: Rebirth	2014	14.99	1,688,924 - 1,757,790	86	Edmund McMillen, Nicalis	Nicalis	Windows, Linux, Mac, PS4, XOne, PSVita, Wii U, New N3DS, iOS	Dedicated engine (C++)	Top-down, dual-stick, action, dungeon crawler	Dungeons, Chistendom, pop culture, horror	2D, pixel	http://bindingofisaacrebirth.gamepedia.com/ ; http://bindingofisaacrebirth.gamepedia.ro m/
Crypt of the Necrodancer	2015	14.99	641,252 - 684,018	87	Brace Yourself Games	Klei Entertainment	Windows, Linux, Mac, PS4, PSVita, iOS	Monkey X	Rhythm, Top-down	Medieval, fantasy	2D, pixel	http://crypt-of-the-necrodancer.wikia.com/
Darkest Dungeon	2016	22.99	1,073,348 - 1,128,438	84	Red Hook Studios	Red Hook Studios	Windows, Linux, Mac, PS4, PSVita	Dedicated engine	Turn-based combat, RPG, dungeon crawler	Fantasy, Lovecraft style, gothic	2D, cartoon	http://darkestdungeon.gamepedia.com/
Don't Starve	2013	14.99	3,925,829 - 4,030,133	79	Klei Entertainment	Klei Entertainment	Windows, Linux, Mac, PS4, PSVita, PS3, XOne, Wii U, iOS, Android	Dedicated engine (C++)	Survival, open world, top-down, adventure, action	Forests	2D, Tim Burton's style	http://dontstarve.wikia.com/
Downwell	2015	2.99	201,727 - 226,037	81	Moppin	Devolver Digital	Windows, PS4, PSVita, iOS, Android	GameMaker	Platforms, action, shoot'em up	Retro	2D, pixel, 3 colors	http://downwell.wikia.com/
Dungeon of the Endless	2014	12.99	595,867 - 637,119	79	AMPLITUDE Studios	AMPLITUDE Studios	Windows, Mac, XOne, iOS	Unity	RPG, dungeon crawler, tower defense	Future, unexplored planet	2D, cartoon	http://dungeon-of-the-endless.wikia.com/
Enter the Gungeon	2016	14.99	526,867 - 565,703	84	Dodge Roll	Devolver Digital	Windows, Linux, Mac, PS4	Unity	Top-down, dual-stick, bullet hell	Medieval, pop culture	2D, pixel	http://enterthegungeon.gamepedia.com/
FTL: Faster Than Light	2012	9.99	2,591,018 - 2,676,044	84	Subset Games	Subset Games	Windows, Linux, Mac, iOS	Dedicated engine (C++)	RPG, Strategy	Space travels	2D, cartoon	http://ftl.wikia.com/ ; http://ftlwiki.com/
Hearth & Slash	2016	22.99	7,960 - 13,392	Not enough critics	AHEARTFULOFGAMES	Badland Games	Windows, Linux, Mac, PS4, XOne	Unity	Hack and slash, beat'em up, action	Robotic future	3D, cartoon	http://heartandslash.gamepedia.com/
Invisible Inc.	2015	19.99	281,910 - 310,516	82	Klei Entertainment	Klei Entertainment	Windows, Linux, Mac, PS4	Moai	Turn-based strategy, tactic, stealth	Cyberpunk	2D, cartoon	http://invisibleinc.wikia.com/ ; http://invisibleinc.gamepedia.com/
Legend of Dungeon	2013	9.99	205,067 - 229,573	62	Robot Loves Kitty	Robot Loves Kitty	Windows, Linux, Mac	Unity	Action, RPG, beat'em up, dungeon crawler	Dungeons, pop culture	2D scenarios (voxel), 2D characters (pixel)	http://legend-of-dungeon.wikia.com/
Luftrausers	2014	9.99	358,127 - 390,277	80	Vlambeer	Devolver Digital	Windows, Linux, Mac, PS3, PSVita, Android	GameMaker	Arcade, action, bullet hell, shooter	Nazy Germany	2D, pixel	http://luftrausers.wikia.com/
Neon Chrome	2016	14.99	7,802 - 13,188	74	10ons	10ons	Windows, Linux, Mac, PS4, XOne	Dedicated engine	Top-down, dual-stick, shooter, action	Cyberpunk	3D, cartoon	http://neonchrome.gamepedia.com/
Nuclear Throne	2015	12.99	362,554 - 394,888	88	Vlambeer	Vlambeer	Windows, Linux, Mac, PS4, XOne, PSVita	GameMaker	Top-down, dual-stick, shooter, bullet hell	Post-apocalyptic	2D, pixel	http://nuclear-throne.wikia.com/
Risk of Rain	2013	9.99	1,374,500 - 1,436,722	77	Hopoo Games	Chucklefish	Windows, Linux, Mac, PS3, PSVita	Dedicated engine	Platforms, dual-stick, shooter, action	Future, unexplored planet	2D, pixel	http://riskofrain.wikia.com/
Rogue Legacy	2013	14.99	1,240,266 - 1,308,628	85	Cellar Door Games	Cellar Door Games	Windows, Linux, Mac, PS4, PSVita, XOne, PS3	XNA	Platforms	Medieval	2D, cartoon	http://rogue-legacy.wikia.com/
Spelunky	2008	14.99	785,579 - 832,829	90	Derek Yu, Mossmouth	Mossmouth	Windows, Mac, PS4, PSVita, PS3, X360	GameMaker	Platforms	Tombs	2D, cartoon	http://spelunky.wikia.com/
Sunless Sea	2015	18.99	419,445 - 454,177	81	Failbetter Games	Failbetter Games	Windows, Linux, Mac	Unity	Survival, exploration	Sea, steampunk, Lovecraft	2D, realistic cartoon	http://sunlesssea.gamepedia.com/
Teleglitch	2013	12.99	764,184 - 810,796	84	Tesi3 Projects	Paradox Interactive	Windows, Linux, Mac	Dedicated engine (C++)	Action, Top-down shooter, survival	Zombies	2D (but 3D scenarios), pixel	http://teleglitch.wikia.com/
This War of Mine	2014	18.99	1,715,439 - 1,784,837	83	11 bit studios	11 bit studios	Windows, Linux, Mac, PS4, XOne, iOS, Android	Dedicated engine	Survival	War	3D (but lateral scroll), realistic	http://this-war-of-mine.wikia.com/ ; http://thiswarofmine.gamepedia.com/
Necropolis	2016	27.99	75,557 - 90,707	59	Harebrained Schemes	Bandai Namco Games	Windows, Mac, PS4, XOne	Unity	Action, dungeon crawler, hack and slash, RPG	Fantasy	3D, minimalist	http://necropolis.gamepedia.com/
Hand of Fate	2015	22.99	343,004 - 374,462	78	Defiant Development	Defiant Development	Windows, Linux, Mac, PS4, XOne	Unity	Action, dungeon crawler, beat em up, cards	Medieval, fantasy	3D, realistic	http://handoffate.gamepedia.com/

Nº characters	Nº Weapons	Nº Items	Nº stages	Nº enemies	Nº bosses	Continuity	Decisions	Character stats
15	1 with accumulable modifications	196	3 chapters with 6 levels the first run, until 6 chapters with 17 levels + extra levels	77	78 including bosses and mini-bosses	Unlockable characters, story continues, more item-spawns	Explore more rooms, swap objects, get new habilities, enter special rooms, open chests, buy items	Health, speed, tears, damage, range, shot speed, luck, keys, bombs
11	1 with accumulable modifications	341	3 chapters with 6 levels the first run, until 6 chapters with a variable number of levels	167	99, including bosses and mini-bosses and variations	Unlockable characters, story continues, more item-spawns	Explore more rooms, swap objects, get new habilities, enter special rooms, open chests, buy items	Health, speed, tears, damage, range, shot speed, luck, keys, bombs
10	18	3 tools + 8 torches + 9 armors + 10 headwear + 10 footwear + 15 rings + 16 consumables + 21 pickups + 6 spells + scenario items	4 zones which 4 floors per zone + boss level	95	7 bosses + 13 minibosses	Player can buy items using diamonds obtained in the last run, unlockable characters, zones, permanent stat upgrades, lowrge storgrate of items	Management of weapons	Health, gold, diamonds
15 classes	Basic weapon + 4 upgrades for each class	Around 88	6 locations	Around 73	15	Improvements on town. Stored characters are maintained	Choose dungeon, buy and use items, manage characters status, dungeon exploration	Accuracy modifier, critical chance, damage, dodge, protection, speed, Health, Stress
9	12	185	19 biomes	68	4	Experience is earned, unlockable characters	Crafting system	Sanity, health, hunger
1	9	21 upgrades	Non-ending water well, 7 styles, 37 palettes	18	1	Unlockable styles and palettes	Reload ammo falling over enemies or touching floor, buy weapons	Health, ammo, crystals
26 + 8 ships	13 melee + 11 ranged	15 suits + 54 devices	12	19	0	Album with collectibles	Map exploration, rooms management, buy items	Health, industry, science, food, energy
7	205	233	5 + extra levels	96	17	Player obtains credits after boat bosses to purchase vehicles, weapons, unlockable characters	Explore more rooms, swap objects, get new habilities, open chests	Health, ammo, keys, bullets, credits
8 races + 10 ships	45	17 systems that can be upgraded + 32 augmentations + 11 drones	5 sectors + 91 events	48	1	Unlockable ships	Manage energy of the ship and crew, choose a route, go to events or not	Health, piloting, engines, shields, weapons, repair, combat
6	75	60 body parts	3 worlds	6	5	Unlockable characters and objects	Choose the equipment	Health, experience points
18	2 melee + 11 ranged + 8 special	6 items + 29 augments + 20 daemons + 27 gadgets + 48 prog ams	9 missions	29	A final mission	Characters can be recovered	Choose missions and spies, upgrade skills, buy items	Helath, action points, power, credits
10 classes	33 melee + 10 ranged + 7 extra weapons + 16 spellbooks	49 hats + 12 enhancements + 3 consumables	26 floors	52	7	Unlockable classes	Map exploration, get and use equipment	Health, damage, defense, experience, gold
Interchangeable parts	7	7 bodies + 7 engines	1	8	1	Unlockable parts	Choose weapon, body and engine, complete missions	Health, shoot interval, speed, damage, bullets, recoil, healing speed, acceleration, turn speed, gravity
5 roles	23 + 17 cybernetic abilities	50 cybernetic enhancements	30	20	5	Perks and stats are maintained, unlockable weapons and perks	Choose weapons and perks, scenario exploration	Health, crits, damage, energy, enhancement slots, luck, speed, rate of fire
12	83 ranged weapons + 11 melee weapons + 6 special weapons	Health, ammo, chests and 12 crowns	7 areas with 15 levels + extra areas	38	4	Unlockable characters, weapons and crowns	Swap weapons, find strategic places, use shortcuts	Health, rad level, 5 types of ammo
12	One weapon (4 attack) per character + additional attacks depending on items	102 items + 10 artifacts + 7 drones	11	27	14	Unlockable characters and items	Choose best path to kill all enemies and reach the teleporter	Health, gold, experience
10 initial classes	1 sword (15 materials) + 11 spells	33 traits + helm, chest, ranger and cape (15 materials) + 11 runes + 32 upgrades	4 areas	83	6	The next generation depends on player actions, gold is maintained	Choose the next generation, scenario exploration	Health, armor, damage, strength, intelligence, gold
20	5 melee + 9 ranged + most of items are throwable	41	4 worlds (4 levels per world except last) + extra levels	Around 36	2 + 11 mini-bosses	Unlockable characters	Choose best path to reach the end door. Buy items in shop	Health, ropes, bombs
46 crew members + 9 ships + 45 officers	The own ship, 9 ships with different liabilities	35 cargo items + 107 curiosities + 46 ship equipment	76 locations	41	At least 3 (included in monsters)	An item, liability or officer can be legacy for the next run	Every event has options, management of resources, crew, missions and map exploration	Iron, mirrors, veils, hearts, pages, ship weight, hold, quarters, hull, terror, hungy, fuel
1	34	31	10	14	3	Unlockable checkpoints	Choose weapons and items	Health, armor
12	5	57	26	Not defined, other NPC with randomized stats	0	Unlockable stories and characters	Home and character management, near locations exploration	Health, hungry
3	105	28 armor pieces + 19 shields + 43 items + 20 codexes + 26 dyes	5 locations	24	0	Unlockable codexes and dyes	Buy items, choose equipment	Health, stamina, gems, tokens
1	25	29 artifacts + 9 armors + 8 shields + 6 gloves + 30 helms + 7 dragon relics + 17 blessings + 28 curses + 35 trinkets	Story mode with quests + endless mode	9	12	Unlockable cards	Choose and buy cards	Health, food, gold

Appendix B

Game statistics

B.1 Enemies

Enemy	Score	Worlds				Health	Melee defense	Magic defense	Speed	Attacks	
Spring	10	x	x			50	0	0	3	10 (Melee)	
SpringSlow	15			x	x	100	0	0.4	1.5	40 (Melee)	
Pump	12	x				20	0.4	0	3	1/particle (Fire)	
PumpBig	20		x	x	x	100	0.8	0	1.5	3/particle (Fire)	
Bounce	10	x	x			30	0	0.1	[12,18]	15 (Contact)	
BounceShot	15		x	x		60	0	0.2	[12,18]	30 (Contact)	15 (Shot)
BounceMultiShot	25			x	x	150	0	0.3	[12,18]	50 (Contact)	30 (MultiShot)
LigthbombKicker	5	x	x			40	0.2	0	5	20 (Kick)	
LigthbombShooter	10	x	x	x		60	0	0.2	5	20 (Shot)	
LigthbombExploder	15			x	x	180	0.3	0	5	40 (Shot) / 30 (kick)	120 (Explosion)
LigthbombComplete	30			x	x	250	0.3	0.3	5	60 (Shot)	
LongArmsBerserk	15	x	x	x		60	0	0.1	3	20 (Lunge)	
LongArmsSpinner	15	x	x	x	x	70	0	0.2	3	25 (Spin)	
LongArmsPursuer	20		x	x	x	120	0	0.3	3	40 (Direct lunge)	
LongArmsComplete	30				x	160	0	0.4	3	70 (Lunge)	60 (Spin)
LongArmsPerfect	40				x	250	0	0.5	3	90 (Direct lunge)	80 (Spin)
SoldierSniper	15	x				60	0	0	0	2/frame (Continuous shot)	
SoldierSniperOneShot	20	x	x	x	x	40	0	0.2	0	25 (Single shot)	
SoldierSniperRotate	30		x	x	x	150	0	0.4	0	4/frame (Continuous shot)	
SoldierSniperRotateSpinner	30			x	x	170	0.3	0.2	0	100 (Spin shot)	

Figure B.1: Enemy statistics

	World 1	World 2	World 3	World 4
Nº enemies per world	9	11	13	12
Score per world	112	170	260	290
Medium score per enemy	12.44444	15.45455	20	24.16667
Expected enemies per room	5	7	10	15
Max score per world	62.22222	108.1818	200	362.5

Figure B.2: World scores depending on the enemies' scores

B.2 Weapons

B.2.1 Melee

Weapon	Tier	Damage	Hits	Animation time	Anim. speed	Real anim. time	Hit time (frames)	Damage radius	Stamina	Animation	Debug key (right control + keypad key)
Basic Axe	1	20	3	3.633	3	1.211	40, 85, 135	1	49.55	one_hand_club_combo	2
Basic Sword	1	30	1	0.833	1.3	0.641	10	1.3	46.82	KnightAttack	1
Long Axe	1	25	2	2.167	1.3	1.667	62, 116	3	30.00	great_sword_jump_attack	+
Iron Hammer	2	50	1	2.95	1.8	1.639	104	0.55	30.51	sword_and_shield_casting	0
Gear Axe	2	30	3	3.533	1.7	2.078	44, 88, 152	2.33	43.31	sword_and_shield_slash_1	6
Steam Hammer	2	60	1	1.3	1.5	0.867	50	0.83	69.23	sword_and_shield_attack_2	9
Steam Axe	2	80	1	2.483	1.3	1.910	60	2.5	41.88	standing_melee_attack_360_low	8
Gold Long Axe	3	70	1	1.3	1	1.300	40	4	53.85	sword_and_shield_attack_1	-
Reforged Blade	3	100	1	0.833	0.5	1.666	10	4.18	60.02	KnightAttack	*
Chainsaw Sword	3	120	2	1.3	1	1.300	10, 33	4.27	184.62	KnightAttack	4
Long Knife	4	150	2	2.733	1.4	1.952	65, 104	4.4	153.68	standing_melee_combo_attack_ver._3	5
Cloud Sword	4	160	1	1.3	1	1.300	40	5.28	123.08	sword_and_shield_attack_1	7
Curved Sword	5	200	2	2.733	2	1.367	65, 104	6.55	292.72	standing_melee_combo_attack_ver._4	3

Figure B.3: Ingame statistics for melee weapons

Id	Price	Health	Stamina	Stamina increase	Melee attack	Melee defense	Magic attack	Magic defense	Speed	Reduction	Tier	Name id
1000	6.960	0.03	0.03	0.01	0.02	0.02	0.03	0.01	0.01	0.0200	1	basic_sword
1001	6.484	0.03	0.02	0.02	0.03	0.03	0.02	0.03	0.01	0.0238	1	basic_axe
1002	31.800	0.1	0.11	0.11	0.11	0.09	0.1	0.09	0.09	0.1000	4	chainsaw_sword
1003	49.118	0.12	0.15	0.15	0.13	0.15	0.14	0.12	0.15	0.1388	5	cloud_sword
1004	44.888	0.12	0.15	0.15	0.12	0.13	0.14	0.12	0.12	0.1313	5	curved_sword
1005	12.255	0.04	0.06	0.03	0.05	0.03	0.06	0.06	0.05	0.0475	2	gear_axe
1006	20.306	0.09	0.09	0.06	0.06	0.07	0.06	0.08	0.06	0.0713	3	gold_long_axe
1007	12.540	0.06	0.06	0.05	0.03	0.05	0.05	0.05	0.03	0.0475	2	iron_hammer
1008	7.628	0.03	0.03	0.02	0.03	0.02	0.02	0.01	0.02	0.0225	1	long_axe
1009	29.828	0.12	0.1	0.09	0.11	0.09	0.11	0.11	0.09	0.1025	4	long_knife
1010	24.248	0.09	0.08	0.06	0.08	0.08	0.07	0.07	0.08	0.0763	3	reforged_blade
1011	15.795	0.05	0.04	0.06	0.05	0.03	0.06	0.06	0.04	0.0488	2	steam_axe
1012	15.930	0.03	0.05	0.03	0.06	0.05	0.04	0.04	0.06	0.0450	2	steam_hammer

Figure B.4: Item statistics for melee weapons

B.2.2 Magic

Weapon	Tier	Damage	Projectiles	Animation time	Anim. speed	Real anim. time	Projectile radius	Projectile speed	Stamina	Debug key (keypad key)
Firebolt	1	10	1	1.333	10	0.133	0.3	25	10	1
Firebolt2	2	20	2	1.333	1	1.333	0.3	25	20	2
Boom	3	100	1	1.333	6	0.222	0.3	25	60	4
Explosion	4	150	1	1.333	6	0.222	0.3	25	100	5
Big Bang	5	200	1	1.333	3	0.444	0.3	5	150	6
Flamethrower	3	2	Flames	1.333	1	1.333	0.01	20	30	3
Groundlight	1	3	1	1.333	4	0.333	2	5	20	7
Groundlight2	3	5	1	1.333	4	0.333	2	30	50	8
Groundlight3	4	7	1	1.333	4	0.333	2	50	100	9
Firebolt3	3	30	3	1.333	1.2	1.111	0.3	25	40	0
Flamethrower2	2	4	Flames x 2	1.333	0.5	2.666	0.01	20	50	+
FastBalls	1	10	5	1.333	20	0.067	0.1	40	10	-
FastBalls2	4	20	7	1.333	20	0.067	0.1	40	40	*

Figure B.5: Ingame statistics for magic spells

Id	Price	Health	Stamina	Stamina increase	Melee attack	Melee defense	Magic attack	Magic defense	Speed	Reduction	Tier	Name id
1000	6.960	0.03	0.03	0.01	0.02	0.02	0.03	0.01	0.01	0.0200	1	basic_sword
1001	6.484	0.03	0.02	0.02	0.03	0.03	0.02	0.03	0.01	0.0238	1	basic_axe
1002	31.800	0.1	0.11	0.11	0.11	0.09	0.1	0.09	0.09	0.1000	4	chainsaw_sword
1003	49.118	0.12	0.15	0.15	0.13	0.15	0.14	0.12	0.15	0.1388	5	cloud_sword
1004	44.888	0.12	0.15	0.15	0.12	0.13	0.14	0.12	0.12	0.1313	5	curved_sword
1005	12.255	0.04	0.06	0.03	0.05	0.03	0.06	0.06	0.05	0.0475	2	gear_axe
1006	20.306	0.09	0.09	0.06	0.06	0.07	0.06	0.08	0.06	0.0713	3	gold_long_axe
1007	12.540	0.06	0.06	0.05	0.03	0.05	0.05	0.05	0.03	0.0475	2	iron_hammer
1008	7.628	0.03	0.03	0.02	0.03	0.02	0.02	0.01	0.02	0.0225	1	long_axe
1009	29.828	0.12	0.1	0.09	0.11	0.09	0.11	0.11	0.09	0.1025	4	long_knife
1010	24.248	0.09	0.08	0.06	0.08	0.08	0.07	0.07	0.08	0.0763	3	reforged_blade
1011	15.795	0.05	0.04	0.06	0.05	0.03	0.06	0.06	0.04	0.0488	2	steam_axe
1012	15.930	0.03	0.05	0.03	0.06	0.05	0.04	0.04	0.06	0.0450	2	steam_hammer

Figure B.6: Item statistics for magic spells

Appendix C

Used assets

Asset	Author	Description	Used in	Source
Procedural Gear	Proroc	Creates customizable gears and worm gears	Main menu details	https://goo.gl/HYWIhG
RAIN AI for Unity	Rival Theory	AI module with pathfinding, routines and decision making	Generation of runtime pathfinding, find routes for enemies and perform actions	https://goo.gl/bz8Y19
3D Games Effects Pack Free	Creepy Cat	8 particle system effects	Projectiles	https://goo.gl/9XCfES
Audio Peer (with custom implementation)	Peer Play	Detects the rhythm of the music	Final bosses and their scenarios	https://goo.gl/NPKsYd
Axe and Cart	naresh	Textured and modeled axe and cart	Used axe as melee weapon	https://goo.gl/xvf32J
BasicBeamShot Free	BeamMan	A shot and continuous laser	The laser of Soldier Snipers	https://goo.gl/fkd6Wu

Book - Page Curl	Abdullah Aldandarawy	A 2D book with interactable pages	Story and controls books	https://goo.gl/xXffzA
Cursor Control	znebby	A driver to manage the mouse using code	To set the position of the cursor for gamepad	https://goo.gl/ia6iBe
PCGDungeons	Nathan Williams	Implementation of three techniques to create random maps	Integrated Delaunay Triangulation for the map generation	https://goo.gl/nXnBKd
Elemental Free	G.E.TeamDev	53 particle systems about water, fire, wind, thunder and so on	Projectiles	https://goo.gl/1boNiz
FT Free Sample	FlyingTeapot	Particles effects about explosions, portals and projectiles	Projectiles	https://goo.gl/ijGrk4
Survival Shooter	Unity Technologies	A top-down game demonstration	Adapted the shooting and cursor systems	https://goo.gl/7JKuVx
KY Magic Effects Free	Kakky	10 particle systems about explosions and projectiles	Projectiles	https://goo.gl/CDVpmB
L20n	Glen De Cauwsemaecker	A module to change automatically the language of the game	All the texts in the game	https://goo.gl/pkAyLy

Lightning 2D/3D	V.Dev	To make lightnings	Lightning Mask boss	https://goo.gl/6nYf3W
Low Poly Fantasy Sword	Club Foot	A textured and modeled sword	The starting melee weapon	https://goo.gl/4PAAd3
TELEPORTER	M31	A teleported made of particle systems	Final portal	https://goo.gl/cSd7x5
Melee Weapon Trail	Anomalous Underdog	Provides a colored trail to attach to objects	All melee weapons use them	https://goo.gl/yoAcWr
Moments	Chman	A continuous screen recording tool	Replays	https://goo.gl/9WFPBf
More Post-Processing Effects	Maxime Jumelle	Screen effects and shaders	Graphical effects	https://goo.gl/u96EGe
Pipes Kit	mojo-structure	Textured and modeled pipes, combin, bending and connections	Obstacles	https://goo.gl/Ay162m
Particle Collection SKJ 2016_Free samples	SKJ	Lot of particle systems about magic	Portals and projectiles	https://goo.gl/PHLWQQ
Fire & Spell Effects	Digital Ruby (Jeff Johnson)	Fire particle systems and a throw system to launch them	Magic and integrated in the combat system	https://goo.gl/9DccLG
Quick Cutscene Creator	Gadget Games	Cutscene editor	Final boss cinematics	https://goo.gl/DsGBjX

White Smoke Particle System	Xenosmash Games	White smoke particle system	Jasper's arm and Training Train	https://goo.gl/YtKQtP
Warrior Pack Bundle 2 FREE	Explosive	Controllable characters with animations	Jasper's animations and animation state machine	https://goo.gl/VEWcZ2
Mixamo	Adobe	High quality animations and characters	Jasper's model and melee animations	https://goo.gl/Mk92GJ
XInputDotNet	speps	Wrapper to manage XInput functions	Xbox and Ps4 gamepads integration	https://goo.gl/fySd3Z
Health Script	YounGen Tech	Health system	Health bars over enemies	https://goo.gl/VVZ5Tk

Appendix D

Used music

Song	Author	Source
My Lane	Lvel 25 Tidehunter	https://goo.gl/IXiER1
DYNAMITE Winamp 5.0RC8 crk	300ix	https://goo.gl/9ONXQH
Unreal Superhero 3 Remix Mashup	Joe Orland	https://goo.gl/wpM9TQ
Mirai Nikki 8-bit NES Remix	TheMusikage	https://goo.gl/8H7Bym
Epic Chiptune	Abdul's Rucksack	https://goo.gl/1noccy
Escape from the City - 8-bit (famitracker)	Squadaloo	https://goo.gl/iffxpm
Attack on Titan Season 2 Opening 8-bit Cover	Musikage	https://goo.gl/RLTQ7X
Sonic Heroes - Seaside Hill [8-Bit Cover]	DanRock Productions	https://goo.gl/fsSo1f
Wake up!/ONE PIECE 8bit	Studio Megaane	https://goo.gl/niSwCe
We are!/ONE PIECE 8bit	Studio Megaane	https://goo.gl/gcdG24
The Hero!! Set Fire to the Furious Fist/One-Punch Man 8bit	Studio Megaane	https://goo.gl/JweFDF
Dynamic!/Dragon Ball Super 8bit	Studio Megaane	https://goo.gl/xVWpQR

Believe/ONE PIECE 8bit	Studio Megaane	https://goo.gl/erjvKR
99/Mob Psycho 100 8bit	Studio Megaane	https://goo.gl/gUC3dZ
Wings of Freedom/Attack on Titan 8bit	Studio Megaane	https://goo.gl/nnz6mr
Thunderstruck/AC/DC 8bit	Studio Megaane	https://goo.gl/2bSmCk
Beautiful World/Evangelion: 1.0 You Are (Not) Alone. 8bit	Studio Megaane	https://goo.gl/CbjjKB
Megalovania - Undertale (8 Bit Universe Version)	8 Bit Universe	https://goo.gl/ZqxzJH
Spider Dance (8 Bit Remix Cover Version) [Tribute to Undertale]	8 Bit Universe	https://goo.gl/EmvntR
Bonetrousle (8 Bit Remix Cover Version) [Tribute to Undertale]	8 Bit Universe	https://goo.gl/A1q8qw
Spear of Justice (8 Bit Remix Cover Version) [Tribute to Undertale]	8 Bit Universe	https://goo.gl/aZ6Ydt
Flyers 8 Bit- Death Parade OP	SnedHlepPls	https://goo.gl/LVGd2f
Keygen Music - 013	Video Game Music Vault	https://goo.gl/mwrsLk
Keygen Music - 012	Video Game Music Vault	https://goo.gl/89Z4XA
Keygen Music - 001	Video Game Music Vault	https://goo.gl/vbL3T3
Keygen Music - 143	Video Game Music Vault	https://goo.gl/SeqH3Z
Keygen Music - 142	Video Game Music Vault	https://goo.gl/iDXUbD
Keygen Music - 114	Video Game Music Vault	https://goo.gl/sj9A6H
Keygen Music - 109	Video Game Music Vault	https://goo.gl/MxtCT9
NARCiSSUS - Offline Explorer Enterprise [Keygen Muisc]	Video Game Music Vault	https://goo.gl/BgizPM
Dedicate your Heart!/Attack on Titan 8bit	Studio Megaane	https://goo.gl/eG6JBs

Give a reason/Slayers NEXT 8bit	Studio Megaane	https://goo.gl/qUPy1k
Opus	Eric Prydz	https://goo.gl/qUPy1k